Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп’ютерних наук

Допущено до захисту
Завідувач кафедри комп’ютерних наук
доктор технічних наук, професор
_____ Андрій БОНДАРЧУК
« ___ » _____ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «бакалавр»
спеціальність 122 Комп’ютерні науки
освітня програма 122.00.01 Інформатика

Тема: ІНТЕРАКТИВНИЙ ПРОГРАМНИЙ ДОДАТОК УПРАВЛІННЯ ЦИФРОВИМИ ЗОБРАЖЕННЯМИ У МОБІЛЬНОМУ СЕРЕДОВИЩІ ANDROID

Виконала
студентка групи ІНб-1-22-4.0д
Буренко Лілія Олегівна
_____
(підпис)

Науковий керівник
кандидат технічних наук,
доцент
Мельник Ірина Юріївна
_____
(підпис)

Київ – 2026

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

«Затверджую»
Завідувач кафедри комп'ютерних наук
доктор технічних наук, професор
_____ Андрій БОНДАРЧУК
« 31 » _____10_____ 2025 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА
«Інтерактивний програмний додаток управління цифровими зображеннями у мобільному середовищі android»

Виконавець – студентка групи ІНб-1-22-4.0д,
спеціальності 122 Комп'ютерні наук,
освітньої програми 122.00.01 Інформатика
Буренко Лілія Олегівна

1. Вихідні дані: Законодавство та нормативно-правові акти України в галузі інформаційних технологій та захисту інтелектуальної власності, наукові публікації та технічна документація з питань розробки мобільних застосунків, управління цифровим контентом, а також офіційна документація до фреймворків React Native та Node.js.
2. Основними завданнями бакалаврської роботи є: проведення огляду літературних джерел, онлайн-публікацій та технічної документації за темою дослідження, аналіз сучасних мобільних застосунків-аналогів для роботи із зображеннями; обґрунтування актуальності, мети, об'єкта, предмета та завдань дослідження; вибір та обґрунтування методів реалізації, включаючи використання засобів react native, node.js sqlite ; проєктування архітектури мобільного застосунку; розробка інтерактивного мобільного застосунку для управління цифровими зображеннями на android з реалізацією функціоналу: завантаження та перегляд зображень, створення та редагування альбомів, налаштування прав доступу до контенту; проведення тестування розробленого застосунку на коректність роботи; оформлення пояснювальної записки згідно з вимогами стандартів та методичних рекомендацій кафедри.
3. Пояснювальна записка: Обсяг – до 96 стор. формату А4 комп'ютерного набору з дотриманням вимог стандарту і методичних рекомендацій кафедри.
4. Графічні матеріали: скриншоти роботи.
5. Додатки: лістинг коду.
6. Строк подання роботи на кафедру: «1» червня 2026 р.

Науковий керівник:
к.т.н., доцент
_____ Мельник І.Ю.
31.10.2025

Виконавець:
_____Буренко Л.О.
31.10.2025

Календарний план

№	Назва етапу дипломної роботи	Строк виконання етапу роботи	Примітка
1	Формування теми дипломної роботи бакалавра	31.10.25	Виконано
2	Ознайомлення з методичними рекомендаціями до дипломної роботи, складання змісту та календарного плану	01.11.25 - 11.01.26	Виконано
3	Написання реферату, вступу та першого розділу	26.01.26 - 15.03.26	Виконано
4	Написання другого розділу та реалізація основного функціоналу	16.03.26 - 31.03.26	Виконано
5	Написання третього розділу та висновків	01.04.26 - 15.04.26	Виконано
6	Виправлення зауважень	16.04.26 - 30.04.26	Виконано
7	Створення презентації	01.05.2026 - 15.05.26	Виконано
8	Захист матеріалів дипломної роботи на засіданні кафедри	12.05.26	Виконано
9	Офіційний захист матеріалів дипломної роботи на засіданні екзаменаційної комісії	15.06.2026	Виконано

«Затверджую»

Науковий керівник
к.т.н., доцент, Мельник І.Ю.

Індивідуальний план склала
Буренко Л.О.

РЕФЕРАТ

Кваліфікаційна робота: 97 с., 18 рисунків, 6 таблиць, 25 посилань.

Актуальність: Стрімке зростання обсягів цифрового фотоконтенту на мобільних пристроях створює потребу в зручних інструментах для його зберігання, організації та керування доступом. Існуючі комерційні рішення мають обмежену гнучкість, залежність від зовнішніх платформ або не забезпечують повного контролю користувача над власними даними.

Об'єкт дослідження: мобільний застосунок для управління цифровими зображеннями користувача, що забезпечує зберігання, організацію, перегляд та взаємодію з візуальним контентом.

Предмет дослідження: методи та засоби реалізації мобільного застосунку для роботи з альбомами, тегами, коментарями та правами доступу до зображень на платформі Android.

Мета роботи: розробка інтерактивного програмного додатку для управління цифровими зображеннями в мобільному середовищі Android, який забезпечує ефективне зберігання, організацію та спільний доступ до графічного контенту.

Завдання:

1. Провести огляд предметної галузі та проаналізувати існуючі рішення у сфері мобільних застосунків для зберігання та керування цифровими зображеннями.
2. Виконати аналіз і обґрунтування вибору технологій та інструментів, необхідних для реалізації функціоналу застосунку.
3. Розробити архітектуру мобільного застосунку, спроєктувати базу даних та реалізувати основні програмні модулі для роботи з цифровими зображеннями.
4. Здійснити перевірку працездатності розробленого застосунку та оцінити відповідність реалізованого функціоналу поставленим вимогам.

Основні результати. Розроблено мобільний застосунок на базі React Native та Node.js із реляційною базою даних SQLite, що реалізує повний цикл

керування цифровими зображеннями. Реалізовано модулі реєстрації та авторизації користувачів, завантаження зображень із метаданими та тегами, організації зображень в альбоми, коментування й оцінювання контенту, а також керування рівнями доступу (публічний, приватний, за посиланням). Проведено тестування застосунку на реальному Android-пристрої та емуляторі, що підтвердило коректність роботи реалізованого функціоналу.

Практичне значення дослідження: полягатиме в практичному використовуванні як основи для побудови персональних медіагалерей, корпоративних систем керування зображеннями або освітніх платформ із підтримкою мультимедійного контенту.

Ключові слова: мобільний застосунок, цифрові зображення, альбоми, JavaScript, Node.js, React Native, Android, SQLite, REST API.

ЗМІСТ

42

ВСТУП

У сучасних умовах стрімкого розвитку цифрових технологій особливого значення набувають інструменти, що дозволяють ефективно працювати з мультимедійною інформацією, зокрема цифровими зображеннями. Щоденне використання смартфонів та інших мобільних пристроїв призводить до постійного зростання обсягів фото- та графічного контенту, який потребує впорядкованого зберігання, зручного перегляду та швидкого доступу. Без належних інструментів управління великі колекції зображень стають складними для навігації, пошуку та подальшого використання. Саме тому актуальним є застосування мобільних рішень, що забезпечують структуровану організацію зображень, їх групування за альбомами, використання тегів, а також можливість коментування та оцінювання контенту. З поширенням мобільного інтернету та хмарних сервісів користувачі очікують від таких застосунків не лише базових функцій перегляду, а й розширених можливостей керування доступом, обміну зображеннями та персоналізації взаємодії з контентом.

Попри наявність значної кількості комерційних сервісів для зберігання фотографій, багато з них мають обмежену гнучкість, залежність від зовнішніх платформ або не забезпечують повного контролю над особистими даними користувача. У зв’язку з цим зростає потреба у створенні альтернативних мобільних застосунків з відкритою архітектурою, локальним або керованим зберіганням даних та можливістю подальшого розширення функціоналу. Такі рішення дозволяють поєднати зручність роботи з цифровими зображеннями, високий рівень конфіденційності та адаптацію під індивідуальні потреби користувачів. У цьому контексті розробка сучасного мобільного застосунку для управління цифровими зображеннями є актуальним та практично значущим завданням.

Це обумовило мету даної кваліфікаційної роботи, яка полягає у розробці мобільного застосунку для управління цифровими зображеннями користувача,

їх зберігання, організації та взаємодії з візуальним контентом. Для досягнення поставленої мети було сформульовано такі завдання:

1. Провести огляд предметної галузі та проаналізувати існуючі рішення у сфері мобільних застосунків для зберігання та керування цифровими зображеннями.

2. Виконати аналіз і обґрунтування вибору технологій та інструментів, необхідних для реалізації функціоналу застосунку управління зображеннями.

3. Розробити архітектуру мобільного застосунку, спроєктувати базу даних та реалізувати основні програмні модулі для роботи з цифровими зображеннями.

4. Здійснити перевірку працездатності розробленого застосунку та оцінити відповідність реалізованого функціоналу поставленим вимогам.

Об’єкт дослідження – мобільний застосунок для управління цифровими зображеннями користувача, що забезпечує зберігання, організацію, перегляд та взаємодію з візуальним контентом.

Предмет дослідження – програмні засоби та технології, які забезпечують розробку мобільного застосунку для зберігання, систематизації та обробки цифрових зображень.

Методи дослідження є загальнонаукові та спеціальні методи аналізу, синтезу, узагальнення, порівняння, проєктування та моделювання, що дали змогу дослідити існуючі рішення у сфері управління цифровими зображеннями, визначити ключові вимоги до функціоналу застосунку, обґрунтувати вибір технологій та забезпечити реалізацію програмної системи відповідно до поставлених завдань.

Практичне значення отриманих результатів полягає в тому, що розроблений мобільний застосунок може використовуватися широким колом користувачів для зручного зберігання та організації цифрових зображень, формування альбомів, роботи з тегами та метаданими, а також ефективної взаємодії з особистими фотоколекціями на мобільних пристроях.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ПРОЄКТОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Введення до предметної галузі

У сучасному цифровому суспільстві візуальна інформація займає одне з провідних місць у процесах комунікації, навчання та збереження особистих даних. Стрімкий розвиток мобільних технологій, а також широке поширення смартфонів із вбудованими камерами високої роздільної здатності призвели до значного зростання обсягів цифрових зображень, що створюються та використовуються щоденно. Фотографії стали не лише засобом фіксації подій, але й важливим елементом соціальної взаємодії, творчого самовираження та інформаційного обміну. Користувачі мобільних пристроїв активно створюють зображення для особистих архівів, професійної діяльності, публікацій у соціальних мережах та обміну з іншими людьми. Внаслідок цього виникає потреба у зручних та ефективних інструментах, які дозволяють не лише зберігати цифрові зображення, але й впорядковувати їх, швидко знаходити потрібні файли, керувати доступом та взаємодіяти з контентом інших користувачів [1].

Особливу актуальність набуває проблема управління великими колекціями зображень у мобільному середовищі. Стандартні галерейні додатки, які постачаються разом з операційними системами мобільних пристроїв, зазвичай забезпечують лише базовий функціонал перегляду та сортування фотографій за датою створення. Однак у випадку значної кількості зображень такі можливості є недостатніми, оскільки не враховують індивідуальні потреби користувача щодо структурування контенту, додавання описів, тегів або створення тематичних альбомів. Крім того, зростає інтерес користувачів до платформ, які поєднують функції управління зображеннями з елементами соціальної взаємодії. Можливість переглядати зображення інших користувачів,

оцінювати їх, залишати коментарі або формувати добірки за інтересами сприяє підвищенню залученості та цінності таких програмних продуктів. Таким чином, системи управління цифровими зображеннями поступово трансформуються з простих сховищ у інтерактивні програмні середовища [2].

Ринок мобільних додатків, пов’язаних із обробкою та зберіганням візуального контенту, активно розвивається протягом останніх років. За даними аналітичних досліджень (рисунок 1.1)[2], категорії мобільних застосунків, пов’язані з фотографією, мультимедіа та соціальною взаємодією, стабільно входять до переліку найбільш популярних у магазинах Google Play та App Store. Це свідчить про підвищений інтерес користувачів до програмних рішень, орієнтованих на роботу з цифровими зображеннями у мобільному середовищі.

Рисунок 1.1. Популярність категорій мобільних додатків, пов’язаних з мультимедійним контентом

Представлена статистика підтверджує зростаючу зацікавленість користувачів у мобільних додатках, орієнтованих на роботу з цифровими зображеннями та мультимедійним контентом. Водночас, незважаючи на наявність великої кількості існуючих рішень, багато з них мають обмежений або надмірно ускладнений функціонал, що ускладнює їх використання для повсякденних потреб.

Таким чином, управління цифровими зображеннями у мобільному середовищі є важливою та актуальною складовою сучасного інформаційного простору. Зростання обсягів візуального контенту, активне використання мобільних пристроїв та підвищення вимог користувачів до зручності роботи з даними обумовлюють значущість дослідження даної предметної галузі. Розгляд особливостей та тенденцій розвитку систем управління цифровими зображеннями дозволяє сформувати цілісне уявлення про сучасний стан та перспективи цього напряму [3].

1.2 Характеристика цифрових зображень як об’єкта обробки

Цифрові зображення є одним з основних видів мультимедійних даних, що активно використовуються у сучасних інформаційних системах, зокрема в мобільних додатках. Вони являють собою двовимірні растрові дані, які складаються з набору пікселів, кожен з яких містить інформацію про колір та яскравість. З огляду на специфіку використання цифрових зображень у програмних системах, доцільно розглянути їх ключові характеристики, що визначають особливості зберігання, обробки та управління такими даними [4].

До основних характеристик цифрових зображень як об’єкта обробки належать:

1. Формат збереження – визначає спосіб кодування графічних даних, рівень стиснення, підтримку прозорості та можливість збереження метаданих. У мобільному середовищі найбільш поширеними є формати JPEG, PNG та WebP, що обумовлено оптимальним співвідношенням між якістю зображення та розміром файлу. Вибір формату безпосередньо впливає на швидкість завантаження зображень, обсяг використовуваної пам’яті та загальну продуктивність мобільного додатку.

2. Метадані зображення – містять додаткову інформацію, що описує властивості файлу та умови його створення. До метаданих належать дата і час створення зображення, параметри зйомки, геолокаційні дані, відомості про авторство, а також технічні характеристики. Використання метаданих

забезпечує можливість систематизації, пошуку та фільтрації зображень, особливо під час роботи з великими обсягами візуального контенту.

3. Назва зображення – є логічним атрибутом, що дозволяє коротко та зрозуміло описати зміст зображення у формі, зручній для користувача. Наявність інформативної назви спрощує навігацію та пошук необхідних зображень у межах фотоколекції.

4. Теги зображення – виконують роль ключових слів або тематичних позначок, за допомогою яких здійснюється групування зображень за певними ознаками. Використання тегів дозволяє реалізувати гнучкі механізми пошуку та впорядкування візуального контенту, що є особливо важливим у разі великої кількості збережених зображень.

5. Спосіб зберігання – визначає місце та умови розміщення цифрових зображень. У мобільних додатках зображення можуть зберігатися як у локальній пам’яті пристрою, так і у віддалених сховищах. Локальне зберігання забезпечує швидкий доступ до файлів без необхідності підключення до мережі, однак обмежується обсягом пам’яті мобільного пристрою. Зберігання зображень у віддалених сховищах дозволяє працювати з великими обсягами даних та забезпечує доступ до контенту з різних пристроїв, проте потребує стабільного мережевого з’єднання. Незалежно від обраного способу зберігання, важливими аспектами залишаються оптимізація обсягу даних, забезпечення швидкого доступу до файлів та збереження цілісності пов’язаних метаданих [5].

Для узагальнення основних характеристик цифрових зображень доцільно розглянути найбільш поширені формати збереження, що використовуються у мобільних додатках (таблиця 1.1).

Таким чином, цифрові зображення як об’єкт обробки характеризуються поєднанням графічної інформації та супровідних даних, що визначають особливості їх зберігання та використання у мобільних додатках. Формати зображень, метадані, назви та теги є ключовими елементами, які впливають на організацію, впорядкування та доступ до візуального контенту. Розуміння

зазначених характеристик дозволяє комплексно описати цифрові зображення з точки зору їх представлення та обробки в сучасних програмних системах [6].

Таблиця 1.1

Основні формати цифрових зображень та їх характеристики

Формат	**Тип стиснення**	**Підтримка прозорості**	**Підтримка метаданих**	**Особливості використання**
JPEG	З втратами	Ні	Так	Малий розмір файлу, широко використовується
PNG	Без втрат	Так	Обмежена	Висока якість, використовується для зображень з прозорістю
WebP	З втратами / без втрат	Так	Так	Сучасний формат з високим рівнем стиснення
BMP	Без стиснення	Ні	Ні	Великий обсяг файлу, рідко використовується в мобільних системах

1.3 Аналіз існуючих програмних рішень

Сфера управління цифровими зображеннями у мобільному середовищі представлена значною кількістю програмних рішень, орієнтованих на зберігання, організацію та перегляд візуального контенту. Такі додатки відрізняються за функціональними можливостями, підходами до структурування зображень та способами взаємодії з користувачем. Частина рішень фокусується на локальному зберіганні фотографій, інші - на хмарній синхронізації або розширених можливостях організації контенту.

Для формування цілісного уявлення про сучасні програмні засоби управління цифровими зображеннями доцільно розглянути найбільш поширені мобільні застосунки, які використовуються широким колом користувачів. Такий розгляд дозволяє охарактеризувати основні підходи до організації зображень, використання метаданих і навігації у межах фотоколекцій.

1.3.1 Аналіз додатку Google Photos

Google Photos (рисунок 1.2) - один із найбільш відомих і поширених мобільних застосунків для управління цифровими зображеннями[18]. Основним призначенням даного додатку є зберігання, організація та перегляд фотографій і

відео, створених користувачем. Google Photos орієнтований на широке коло користувачів, зокрема власників смартфонів, які активно використовують мобільні камери для створення особистих фотоархівів. Додаток поєднує простий інтерфейс із автоматизованими механізмами впорядкування зображень, що робить його зручним для повсякденного використання.

Розглянемо ключовий функціонал даного додатку:

Завантаження та зберігання зображень – додаток дозволяє автоматично або вручну завантажувати фотографії та відео з мобільного пристрою, забезпечуючи їх централізоване зберігання в межах облікового запису користувача.

Автоматична організація зображень – фотографії групуються за датою створення, місцем зйомки та іншими параметрами, що спрощує навігацію у великій колекції зображень.

Створення альбомів – користувач може об’єднувати зображення у тематичні альбоми для зручнішого перегляду та впорядкування контенту.

Пошук зображень – реалізовано пошук фотографій за датою, місцем або іншими характеристиками, що дозволяє швидко знаходити потрібні зображення.

Перегляд та базове редагування – додаток надає можливість переглядати зображення у повноекранному режимі та виконувати базові операції редагування, такі як обрізання або корекція яскравості.

Наступним кроком розглянемо ключові переваги, виявлені користувачами при використанні Google Photos:

1. Простий та зрозумілий інтерфейс – додаток є інтуїтивно зрозумілим навіть для користувачів без технічної підготовки.
2. Автоматизація процесів організації – мінімальна потреба у ручному впорядкуванні зображень завдяки автоматичному групуванню.
3. Доступність з різних пристроїв – користувач може переглядати свої зображення з будь-якого мобільного пристрою після авторизації.

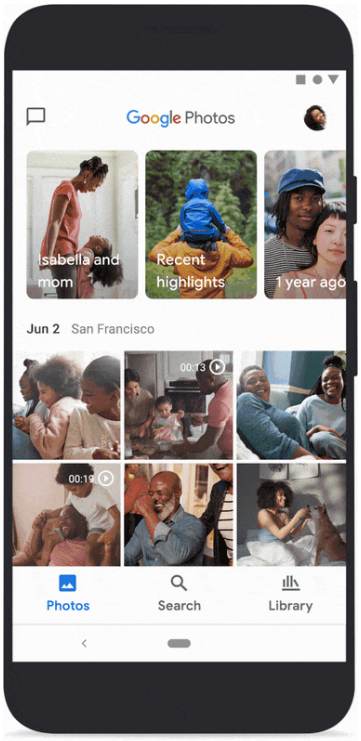

Рисунок 1.2. Інтерфейс користувача Google Photos

Наступним кроком розглянемо ключові недоліки, виявлені користувачами при використанні Google Photos:

1. Обмеження безкоштовного сховища – обсяг доступного місця для зберігання зображень є обмеженим без оформлення платної підписки.

2. Залежність від мережевого з’єднання – для повноцінної роботи з хмарними даними необхідний доступ до інтернету.

3. Обмежені можливості ручного керування – автоматична організація не завжди дозволяє гнучко налаштовувати структуру зберігання зображень відповідно до індивідуальних потреб користувача.

Підсумовуючи, можна зазначити, що Google Photos є популярним та зручним інструментом для управління цифровими зображеннями у мобільному середовищі. Поєднання простоти використання з автоматизованими механізмами організації дозволяє ефективно працювати з великими обсягами візуального контенту. Незважаючи на наявні обмеження, додаток залишається одним із найбільш поширених рішень у сфері управління цифровими зображеннями.

1.3.2 Аналіз додатку Flickr

Flickr (рисунок 1.3) - відомий мобільний застосунок та онлайн-платформа, призначені для зберігання, організації та перегляду цифрових зображень[18]. Основною метою Flickr є надання користувачам зручних інструментів для впорядкування власних фотоколекцій, а також можливості зберігання зображень у структурованому вигляді. Додаток широко використовується як звичайними користувачами, так і фотографами-аматорами, які прагнуть систематизувати свої зображення та забезпечити до них зручний доступ.

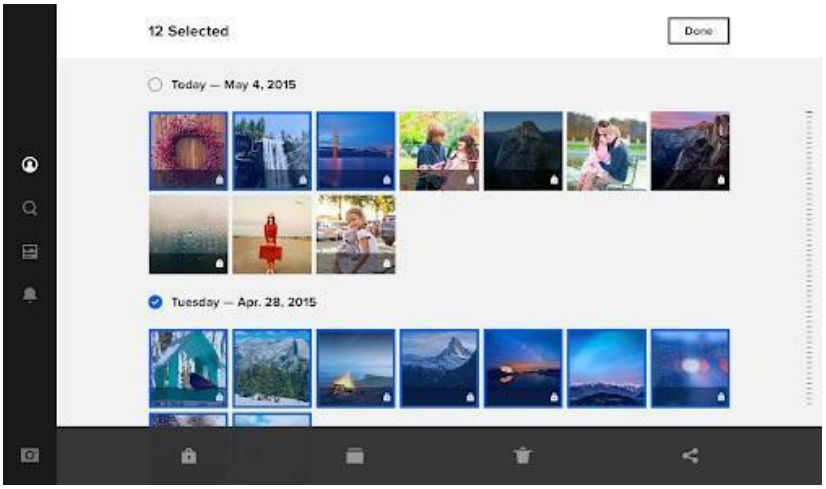

Рисунок 1.3. Інтерфейс користувача Flickr

Розглянемо ключовий функціонал даного додатку:

Завантаження та зберігання зображень – Flickr дозволяє користувачам завантажувати фотографії з мобільного пристрою та зберігати їх у межах власного облікового запису.

Організація зображень за альбомами – користувач може групувати фотографії у тематичні альбоми, що спрощує структурування фотоколекції.

Використання тегів – додаток підтримує додавання тегів до зображень, що дозволяє здійснювати зручний пошук і категоризацію контенту.

Перегляд зображень – реалізовано можливість перегляду фотографій у різних режимах, зокрема у повноекранному форматі.

Керування приватністю – користувач може налаштовувати рівень доступу до зображень, визначаючи, які з них є приватними, а які доступні для перегляду іншими користувачами.

Наступним кроком розглянемо ключові переваги, виявлені користувачами при використанні Flickr:

1. Зручні засоби організації контенту – наявність альбомів і тегів дозволяє ефективно впорядковувати великі обсяги зображень.
2. Висока якість збережених зображень – додаток орієнтований на зберігання фотографій без значної втрати якості.
3. Гнучкі налаштування доступу – можливість керування приватністю забезпечує контроль над власним контентом.

Наступним кроком розглянемо ключові недоліки, виявлені користувачами при використанні Flickr:

1. Обмеження безкоштовної версії – деякі можливості зберігання та управління зображеннями доступні лише у платній підписці.
2. Менш інтуїтивний інтерфейс для нових користувачів – у порівнянні з простішими галерейними додатками, освоєння функціоналу може потребувати додаткового часу.
3. Залежність від інтернет-з’єднання – для доступу до збережених зображень необхідне стабільне підключення до мережі.

Підсумовуючи, можна зазначити, що Flickr є функціональним програмним рішенням для управління цифровими зображеннями у мобільному середовищі. Наявність розвинених механізмів організації, таких як альбоми та теги, дозволяє ефективно працювати з великими фотоколекціями. Незважаючи на певні обмеження, додаток залишається популярним серед користувачів, які потребують розширених можливостей впорядкування зображень.

1.3.3 Аналіз додатку Slidebox

Slidebox (рисунок 1.4) - мобільний застосунок, призначений для впорядкування та керування цифровими зображеннями, що зберігаються у

пам’яті мобільного пристрою[18]. Основною ідеєю Slidebox є надання користувачеві простих і швидких інструментів для сортування фотографій без використання складних налаштувань або хмарних сервісів. Додаток орієнтований на користувачів, які прагнуть навести порядок у власній фотогалереї, видалити зайві зображення та розподілити фотографії за альбомами.

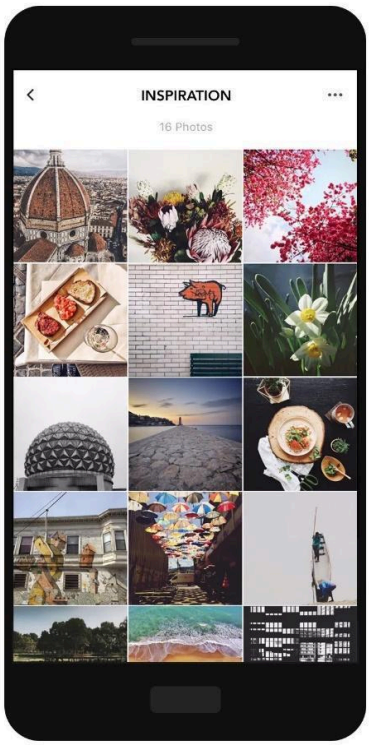

Рисунок 1.4. Інтерфейс користувача Slidebox

Розглянемо ключовий функціонал даного додатку:

Сортування зображень жестами – Slidebox дозволяє переміщувати фотографії між альбомами або видаляти їх за допомогою простих жестів, що значно пришвидшує процес впорядкування.

Створення та керування альбомами – користувач може створювати нові альбоми та додавати до них зображення безпосередньо під час перегляду фотографій.

Перегляд зображень у повноекранному режимі – додаток забезпечує зручний перегляд фотографій із можливістю швидкого переходу між ними.

Видалення непотрібних фотографій – реалізовано простий механізм видалення зображень безпосередньо під час сортування.

Робота з локальним сховищем – усі операції виконуються з фотографіями, що зберігаються у пам’яті пристрою, без необхідності синхронізації з хмарними сервісами.

Наступним кроком розглянемо ключові переваги, виявлені користувачами при використанні Slidebox:

1. Простота та зручність використання – мінімалістичний інтерфейс і керування жестами роблять додаток інтуїтивно зрозумілим.
2. Висока швидкість сортування – можливість швидко переглядати та впорядковувати великі колекції зображень.
3. Відсутність залежності від інтернету – робота з фотографіями не потребує мережевого з’єднання.

Наступним кроком розглянемо ключові недоліки, виявлені користувачами при використанні Slidebox:

1. Обмежений функціонал – додаток не підтримує розширені можливості редагування або пошуку зображень.
2. Відсутність хмарного зберігання – користувач не має можливості синхронізувати зображення між різними пристроями.
3. Орієнтація лише на локальні галереї – додаток працює виключно з фотографіями, що зберігаються на пристрої.

Підсумовуючи, можна зазначити, що Slidebox є простим і ефективним інструментом для управління цифровими зображеннями у мобільному середовищі. Орієнтація на швидке сортування та локальне зберігання робить додаток зручним для користувачів, які прагнуть упорядкувати власну фотогалерею без використання складних сервісів. Обмежений функціонал компенсується простотою та швидкістю роботи, що визначає основне призначення даного програмного рішення.

1.4 Визначення вимог до системи

Проведений аналіз існуючих програмних рішень для управління цифровими зображеннями у мобільному середовищі дозволяє зробити узагальнені висновки щодо їх функціональних можливостей, особливостей використання та обмежень. Розглянуті додатки демонструють різні підходи до організації фотоконтенту - від автоматизованого хмарного зберігання до локального впорядкування зображень за допомогою простих інструментів. Водночас жодне з проаналізованих рішень не може в повній мірі задовольнити всі потреби користувачів, що зумовлює необхідність формування чітких вимог до проєктованої системи.

Серед поширених недоліків існуючих програмних продуктів можна виділити обмежену гнучкість управління зображеннями, залежність від мережевого з’єднання, наявність функціональних обмежень у безкоштовних версіях, а також складність інтерфейсу для користувачів без відповідного досвіду. Частина застосунків орієнтована переважно на автоматичну обробку та хмарне зберігання, що знижує можливості ручного керування контентом, тоді як інші надають лише базовий набір інструментів без розширених механізмів пошуку та систематизації.

Разом із тим аналіз показав наявність ряду переваг, які є спільними для успішних програмних рішень у даній предметній галузі. До таких переваг належать зручний та інтуїтивно зрозумілий інтерфейс, підтримка організації зображень за альбомами та тегами, можливість швидкого пошуку, а також забезпечення надійного зберігання даних. Саме поєднання простоти використання з достатнім рівнем функціональності визначає практичну цінність систем управління цифровими зображеннями для широкого кола користувачів.

Таким чином, результати аналізу існуючих програмних рішень дозволяють сформулювати обґрунтований перелік вимог до системи управління цифровими зображеннями у мобільному середовищі. Визначення таких вимог є необхідним етапом проєктування, оскільки воно забезпечує відповідність

майбутньої системи сучасним очікуванням користувачів та врахування виявлених переваг і недоліків аналогічних програмних продуктів.

1.4.1 Визначення функціональних вимог до системи

Для ефективного проєктування та реалізації інтерактивного програмного додатку управління цифровими зображеннями у мобільному середовищі необхідно чітко визначити його функціональні вимоги. Формалізація таких вимог дозволяє забезпечити узгодженість між очікуваннями користувачів та можливостями програмної системи, а також створює основу для побудови логіки роботи додатку та його інтерфейсу. Чітко сформульовані функціональні вимоги дають змогу однозначно визначити перелік функцій, які повинні бути реалізовані в системі, та забезпечують цілісність її поведінки у різних сценаріях використання [7].

Функціональні вимоги являють собою сукупність характеристик, що описують дії, які повинна виконувати система, її реакцію на дії користувача та правила обробки даних. Вони визначають основні можливості програмного забезпечення, включаючи роботу з користувацькими обліковими записами, управління цифровими зображеннями, організацію контенту та взаємодію з базою даних. Чітке визначення функціональних вимог є важливим етапом проєктування, оскільки саме вони формують перелік функцій, необхідних для повноцінної роботи системи, а також спрощують процес тестування та подальшої підтримки програмного продукту [8]. Крім того, функціональні вимоги слугують основою для оцінки відповідності реалізованого програмного рішення поставленим завданням. Їх формалізація дозволяє забезпечити узгодженість між етапами проєктування, реалізації та впровадження системи.

Функціональні вимоги до проєктованого мобільного додатку наведено у таблиці 1.2.

Таблиця 1.2

Специфікація функціональних вимог до проєктованого мобільного додатку

№	Назва	Пріоритет	Складність	Контакт
1	Реєстрація користувача	Обов’язковий	Середня	Користувач, Система, База даних
2	Авторизація користувача	Обов’язковий	Середня	Користувач, Система, База даних
3	Завершення сеансу користувача	Обов’язковий	Низька	Користувач, Система
4	Завантаження цифрових зображень	Обов’язковий	Середня	Користувач, Система, База даних
5	Перегляд власних зображень	Обов’язковий	Низька	Користувач, Система, База даних
6	Редагування назви та тегів зображень	Обов’язковий	Середня	Користувач, Система, База даних
7	Видалення власних зображень	Обов’язковий	Низька	Користувач, Система, База даних
8	Створення та керування альбомами	Обов’язковий	Середня	Користувач, Система, База даних
9	Перегляд зображень інших користувачів	Обов’язковий	Середня	Користувач, Система, База даних
10	Оцінювання зображень за п’ятибальною шкалою	Бажаний	Низька	Користувач, Система, База даних
11	Пошук зображень за назвою або тегами	Обов’язковий	Середня	Користувач, Система, База даних
12	Сортування та фільтрація зображень	Бажаний	Середня	Користувач, Система, База даних

Функціональні вимоги, наведені у таблиці 1.2, є основою для формування логіки роботи проєктованого програмного додатку. Їх формалізація дозволяє чітко визначити очікувану поведінку системи, спроєктувати відповідні

програмні модулі та забезпечити узгоджену взаємодію між користувацьким інтерфейсом і базою даних.

Урахування зазначених функціональних вимог на етапі проєктування сприяє зменшенню ризиків помилок у реалізації, спрощує процес тестування та забезпечує цілісність програмного продукту. Чітке визначення функціональних можливостей дозволяє створити зручний і зрозумілий мобільний додаток для управління цифровими зображеннями, орієнтований на потреби кінцевих користувачів.

1.4.2 Визначення нефункціональних вимог до системи

Наступним кроком є визначення нефункціональних вимог до системи, які описують загальні характеристики та якісні показники роботи інтерактивного програмного додатку управління цифровими зображеннями у мобільному середовищі. На відміну від функціональних вимог, нефункціональні не визначають конкретні дії системи, а формують умови, у межах яких має реалізовуватись уся функціональність програмного забезпечення. Саме ці вимоги значною мірою впливають на зручність користування, продуктивність, безпеку та стабільність роботи мобільного додатку в реальних умовах експлуатації.

Нефункціональні вимоги є критично важливими, оскільки вони безпосередньо впливають на користувацький досвід. Навіть за умови повної відповідності реалізованого функціоналу очікуванням користувачів, недотримання нефункціональних характеристик, таких як швидкість відгуку, адаптивність інтерфейсу чи захист персональних даних, може суттєво знизити загальну якість системи. Тому врахування нефункціональних вимог на етапі проєктування є необхідною умовою створення надійного та зручного програмного продукту [9].

Нефункціональні вимоги до проєктованого мобільного додатку наведено у таблиці 1.3.

Таблиця 1.3

Специфікація нефункціональних вимог до проєктованого мобільного додатку управління цифровими зображеннями

№	Назва вимоги	Пріоритет	Складність	Контакт
1	Кросплатформеність (Android / iOS)	Обов’язковий	Висока	Система
2	Висока продуктивність та швидкий відгук інтерфейсу	Обов’язковий	Висока	Система
3	Надійність та стабільність роботи	Обов’язковий	Середня	Система
4	Безпека зберігання та обробки персональних даних	Обов’язковий	Висока	Система, База даних
5	Захист механізмів автентифікації користувачів	Обов’язковий	Середня	Система
6	Зручний та інтуїтивно зрозумілий інтерфейс користувача (UI/UX)	Обов’язковий	Висока	Користувач, Система
7	Адаптивність інтерфейсу до різних розмірів екранів	Обов’язковий	Середня	Користувач, Система
8	Масштабованість архітектури системи	Бажаний	Висока	Система
9	Можливість подальшого розширення функціоналу	Бажаний	Висока	Система
10	Відповідність сучасним стандартам розробки ПЗ	Бажаний	Середня	Система

Наведені у таблиці 1.3 нефункціональні вимоги є невід’ємною складовою процесу створення якісного мобільного додатку для управління цифровими зображеннями. Хоча вони не визначають конкретну поведінку системи, їх дотримання безпосередньо впливає на надійність, швидкодію, безпеку та зручність використання програмного продукту. Урахування нефункціональних вимог на етапі проєктування дозволяє побудувати технічно обґрунтовану архітектуру, адаптовану до реальних умов експлуатації, а також зменшити ризики, пов’язані з масштабуванням, оновленням і супроводом системи.

Висновки до розділу 1

У першому розділі було проведено аналіз предметної галузі проєктованого програмного забезпечення, що дозволило сформувати цілісне уявлення про особливості управління цифровими зображеннями у мобільному середовищі. Розглянуто роль цифрових зображень у сучасних інформаційних

системах, їх основні характеристики як об’єкта обробки, а також специфіку зберігання, організації та використання візуального контенту на мобільних пристроях.

У ході аналізу існуючих програмних рішень було встановлено, що сучасні мобільні додатки для управління цифровими зображеннями реалізують різні підходи до організації фотоконтенту - від автоматизованого хмарного зберігання до локального впорядкування зображень із мінімальним набором функцій. Розглянуті застосунки відрізняються за рівнем функціональності, зручністю використання, способами організації даних та залежністю від мережевого з’єднання, що свідчить про відсутність універсального рішення, яке б повністю відповідало всім потребам користувачів.

На основі проведеного аналізу було визначено функціональні та нефункціональні вимоги до системи управління цифровими зображеннями у мобільному середовищі. Функціональні вимоги описують основні можливості системи, пов’язані з роботою користувачів і управлінням зображеннями, тоді як нефункціональні вимоги визначають якісні характеристики програмного продукту, такі як продуктивність, надійність, безпека та зручність користування.

Таким чином, результати аналізу предметної галузі та сформульовані вимоги створюють ґрунтовну основу для подальшого проєктування та реалізації інтерактивного програмного додатку управління цифровими зображеннями у мобільному середовищі з урахуванням сучасних потреб користувачів і особливостей мобільних платформ.

РОЗДІЛ 2

АНАЛІЗ ТА ВИЗНАЧЕННЯ ІНСТРУМЕНТІВ РЕАЛІЗАЦІЇ ІНТЕРАКТИВНОГО ПРОГРАМНОГО ДОДАТКУ УПРАВЛІННЯ ЦИФРОВИМИ ЗОБРАЖЕННЯМИ

2.1 Вибір програмного забезпечення для розробки інтерактивного програмного додатку управління цифровими зображеннями

У сучасному цифровому середовищі програмні додатки для роботи з цифровими зображеннями відіграють важливу роль у зберіганні, перегляді, обробці та організації мультимедійного контенту. Зі зростанням обсягів візуальних даних, що створюються користувачами, актуальним стає питання ефективного управління зображеннями за допомогою зручних та інтуїтивно зрозумілих програмних засобів. Розробка інтерактивного програмного додатку управління цифровими зображеннями потребує ретельного вибору програмного забезпечення, яке забезпечить стабільну роботу, високу швидкодію, зручний інтерфейс користувача та можливість подальшого розширення функціоналу.

На початковому етапі проєктування особливо важливо визначити мову програмування, фреймворк, середовище розробки та допоміжні інструменти, що використовуватимуться для реалізації основних функцій додатку. До таких функцій належать завантаження та зберігання зображень, їх сортування за різними критеріями, перегляд, базова обробка, робота з метаданими, а також взаємодія з локальним або хмарним сховищем даних. Вибрані технології повинні забезпечувати ефективну роботу з графічними даними та коректну обробку великих обсягів інформації.

Окрему увагу доцільно приділити засобам кросплатформенної розробки, які дозволяють створювати програмний додаток, сумісний із різними операційними системами, що розширює коло потенційних користувачів. Для додатків управління цифровими зображеннями важливими є швидка реакція інтерфейсу, можливість офлайн-доступу до збережених даних, підтримка різних

форматів зображень, а також інтеграція зі сторонніми сервісами, зокрема хмарними сховищами або зовнішніми джерелами медіаконтенту. Крім того, значну роль відіграє продуманий дизайн інтерфейсу, простота навігації та можливість персоналізації відображення зображень.

На рисунку 2.1[4] наведено статистику використання мов програмування, які найчастіше застосовуються під час розробки програмних додатків для роботи з цифровими зображеннями та мультимедійними даними [10].

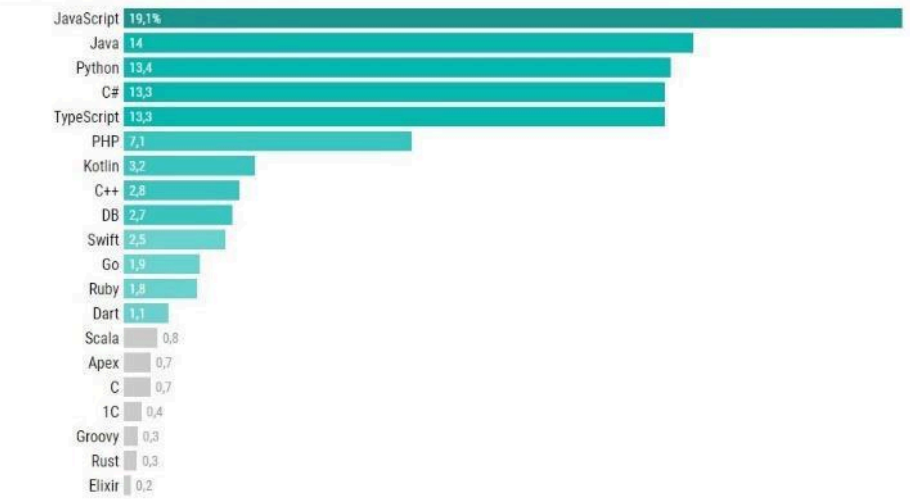

Рисунок 2.1. Статистика використання мов програмування у розробці серверної логіки програмних додатків управління цифровими зображеннями

На основі наведених даних можна зробити висновок, що мова програмування JavaScript на сьогодні є однією з найпоширеніших для реалізації серверної та клієнтської логіки у відповідній предметній галузі. Водночас такі мови програмування, як Python, C# та Java, також широко застосовуються завдяки своїм можливостям у роботі з графічними даними, стабільності та наявності розвинених бібліотек і фреймворків. Використання зазначених мов програмування дозволяє ефективно реалізовувати функціонал обробки, зберігання та візуалізації цифрових зображень.

2.2 Порівняльний аналіз мов програмування для розробки інтерактивного програмного додатку управління цифровими зображеннями

Для аналізу характеристик найбільш актуальних для виконання поставленого завдання технологій доцільно використати метод порівняльного аналізу. З цією метою буде складено низку порівняльних таблиць, що дозволяють наочно зіставити основні властивості обраних мов програмування. Порівняння здійснюватиметься за такими характеристиками, як парадигми програмування, типізація, керування пам’яттю та об’єктно-орієнтовані можливості [11]. Використання табличного подання інформації спрощує процес аналізу та дозволяє зробити узагальнені висновки щодо доцільності застосування відповідних технологій у межах даного програмного проєкту[1].

Таблиця 2.1

Порівняльна характеристика за парадигмами

Можливість	C#	Java	C++	PHP	Python	JS
Імперативна	+	+	+	+	+	+
Обїєктно-орієнтована	+	+	+	+	+	+
Функціональна	+/-	+/-	+/-	+/-	+	+
Рефлексивна	+/-	+/-	+	+	+	+

Таблиця 2.2

Порівняльна характеристика за типізацією

Можливість	C#	Java	C++	PHP		Python	JS
Статична типізація	+	+	-	-		-	-
Дин.типізація	+	-	+	+		+	+
Явна типізація	+/-	+	-	+/-		+/-	-
Неявна типізація	+	-	+	+		+	+
Неявне приведення	-	+	+	+		+	+

Таблиця 2.3

Порівняльна характеристика за можливостями керування пам’яттю

Можливість	C#	Java	C++	PHP	Python	JS
Об’єкти на стеку	+	-	-	-	-	-
Некеровані вказівники	+	-	-	-	-	-
Ручне керування	+	-	-	-	-	-
Збирання сміття	+	+	+	+	+	+

Таблиця 2.4

Порівняльна характеристика за функціональними можливостями

Можливість	C#	Java	C++	PHP	Python	JS
Декларації функцій	-	-	-	-	-	-
First class функції	+	-	+	+	+	+
Анонімні функції	+	+	+	+	+	+
Лексичні замкнення	+	+	+	+	+	+
Часткове застосування	-	-	+	-	+	+

Таблиця 2.5

Порівняльна характеристика за об’єктно – орієнтованими можливостями

Можливість	C#	Java	C++	PHP	Python	JS
Інтерфейси	+	+	+	+	+	-
Мультиметоди	+/-	-	-	-	-	-
Міксини	-	+	+	+	+	+
Перейменування	-	-	-	-	-	-
Множинне спадкування	-	-	-	-	+	-

Особливу увагу було приділено зручності написання і читання коду, що суттєво впливає на швидкість розробки та супровідність програмного продукту. З аналізу видно, що мови Java та C потребують значної кількості коду для реалізації базової логіки, у той час як такі мови, як JavaScript (зокрема з використанням React Native), дозволяють досягати того самого результату з меншими зусиллями завдяки декларативному підходу та наявності великої кількості готових компонентів.

Під час розробки інтерактивного програмного додатку управління цифровими зображеннями важливо враховувати ключові технічні можливості, які забезпечують стабільну роботу системи, зручність взаємодії з користувачем та можливість подальшого розширення функціоналу. До основних вимог таких додатків належать: кросплатформенна підтримка (Android/iOS), локальне збереження цифрових зображень і пов’язаних з ними метаданих, швидка обробка та відображення графічного контенту, оперативне реагування інтерфейсу, підтримка адаптивного дизайну для різних типів пристроїв, а також зручна навігація між елементами медіаконтенту. Усі зазначені вимоги ефективно реалізуються з використанням технологічного стеку JavaScript у поєднанні з React Native. У контексті виконання поставленого завдання даний стек має такі переваги:

1. Створення єдиної кодової бази для Android і iOS, що значно скорочує час і вартість розробки.
2. Активна спільнота та підтримка з боку великої кількості розробників, а також стабільний розвиток технології.
3. Наявність широкого спектра бібліотек для реалізації інтерфейсу та роботи з даними, зокрема засобів навігації, асинхронної взаємодії з API та локального збереження інформації.
4. Проста інтеграція з вебсервісами, хмарними сховищами та зовнішніми API, що є важливим для синхронізації та резервного копіювання цифрових зображень.
5. Великий обсяг навчальних матеріалів і прикладів, що спрощує процес розробки
6. Підтримка механізмів швидкого оновлення інтерфейсу під час розробки, що дозволяє оперативно перевіряти внесені зміни.
7. Розвинена екосистема модулів і плагінів, яка дає змогу легко додавати нові функції, пов’язані з обробкою та відображенням зображень.
8. Інструменти Expo, що спрощують процес тестування, налагодження та збірки програмного додатку для різних платформ.

9. Наявність офіційної документації та практичного досвіду використання даного стеку у подібних програмних рішеннях, що підвищує надійність обраного підходу.

Підсумовуючи, можна зазначити, що хоча кожна мова програмування та фреймворк має власні переваги й обмеження, для розробки інтерактивного програмного додатку управління цифровими зображеннями найбільш доцільним є використання JavaScript у поєднанні з React Native. Даний технологічний стек забезпечує кросплатформенність, гнучкість побудови інтерфейсу, високу швидкість розробки та широкий набір готових інструментів, що робить його оптимальним вибором для створення сучасного програмного додатку для роботи з цифровими зображеннями [12].

2.3 Вибір текстового редактора для написання коду

Для реалізації коду, потрібного для дипломного проєкту, зазвичай використовують текстовий редактор - програму або компонент, що служить для створення та корекції текстових файлів. Редактор є одним із головних інструментів розробника, дозволяючи не тільки писати код, а й відразу його перевіряти і виправляти.

Вибір редактора для роботи над проєктом - непросте завдання. Воно включає аналіз доступних варіантів, врахування власних вподобань, а також практичну перевірку, що допомагає обрати редактор, який найкраще відповідатиме конкретним вимогам завдання.

Sublime Text Editor (Рисунок 2.2)[18] - це потужний та гнучкий текстовий редактор, популярний серед розробників завдяки своїй швидкості, широким можливостям налаштування і підтримці численних мов програмування. Він забезпечує інтуїтивний інтерфейс та функціонал, що робить процес кодування більш зручним і ефективним.

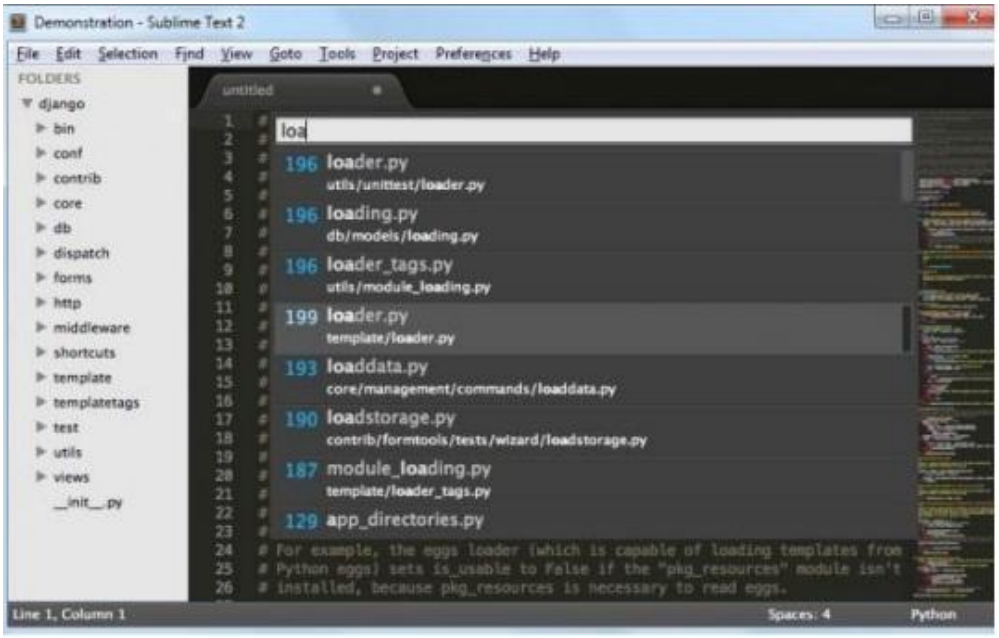

Рисунок 2.2. Інтерфейс користувача Sublime Text editor

Розглянемо функціонал Sublime Text Editor:

Швидкий пошук і навігація - забезпечує можливість швидко знаходити необхідні файли або рядки коду, значно прискорюючи робочий процес.

Режим розділення екрану - дозволяє одночасно працювати з кількома файлами, що дуже зручно при багатозадачності.

Підтримка плагінів - Sublime Text легко розширюється за допомогою плагінів, що додає нові можливості та покращує зручність роботи.

Розширені можливості налаштування - дозволяє користувачам підлаштовувати інтерфейс та функції редактора під свої індивідуальні потреби.

Розглянемо переваги, що були виявлені користувачами при використанні Sublime Text Editor:

1. Швидкодія та мінімальне навантаження на систему - редактор працює дуже швидко навіть на старих комп’ютерах.
2. Інтуїтивний інтерфейс - простий і зрозумілий інтерфейс, який легко налаштувати під себе.

Розглянемо недоліки, що були виявлені користувачами при використанні Sublime Text Editor:

1. Відсутність безкоштовної повної версії -повноцінний доступ вимагає придбання ліцензії.
2. Обмежена підтримка проєктів з великою кількістю файлів - робота може сповільнюватися при відкритті великих проєктів.

Загалом, Sublime Text Editor - це відмінний інструмент для написання коду з багатьма корисними функціями та можливостями для налаштування. Він ідеально підходить для розробників, які шукають швидкий, гнучкий і зручний редактор, хоч деякі його недоліки можуть обмежити використання для великих командних проєктів або проєктів із масштабною файловою структурою [13].

Visual Studio Code (рисунок 2.3)[18] – це безкоштовний і потужний текстовий редактор, створений компанією Microsoft. Він відзначається широким функціоналом, підтримкою різних мов програмування, вбудованими інструментами для розробки та активною спільнотою користувачів. Завдяки своїм можливостям і простоті використання, VS Code здобув популярність серед розробників у всьому світі.

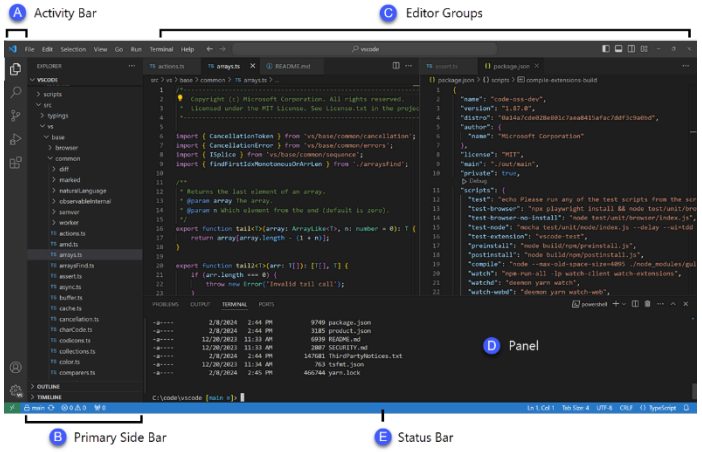

Рисунок 2.3. Інтерфейс користувача Visual Studio Code

Розглянемо функціонал Visual Studio Code:

Інтегрований термінал - дозволяє виконувати командний рядок прямо в редакторі, що значно прискорює процес розробки.

Вбудована підтримка Git - забезпечує зручну роботу з системою контролю версій, дозволяючи відслідковувати зміни та управляти репозиторіями без необхідності виходити з редактора.

Маркетплейс розширень - надає доступ до величезної кількості плагінів, що додають функції для будь-яких потреб, від підсвічування синтаксису до інтеграції з різними сервісами.

Підтримка дебагінгу - дозволяє налагоджувати код безпосередньо в редакторі з допомогою інтегрованих інструментів для багатьох мов програмування.

Розглянемо переваги, що були виявлені користувачами при використанні Visual Studio Code:

1. Безкоштовність та відкритий код - VS Code є повністю безкоштовним та підтримує open-source спільноту, що дозволяє його розширювати та вдосконалювати.
2. Широка підтримка розширень - маркетплейс VS Code пропонує численні плагіни, що дозволяють розширити функціонал редактора для будь-яких задач.

Розглянемо недоліки, що були виявлені користувачами при використанні Visual Studio Code:

1. Високе споживання ресурсів - VS Code може сповільнюватися на комп’ютерах з обмеженими ресурсами через свою функціональність і підтримку розширень.
2. Затримки при роботі з великими проєктами - при відкритті великих проєктів редактор може працювати повільніше, особливо якщо активовано багато розширень.

Visual Studio Code - це універсальний і багатофункціональний редактор для розробки, що підійде як новачкам, так і досвідченим розробникам. Його можливості для налаштування та розширення роблять його ефективним інструментом для широкого спектра задач, хоча високе навантаження на систему може обмежити його використання на менш потужних пристроях [13].

Notepad ++ (Рисунок 2.4)[18] - це безкоштовний текстовий редактор для Windows, який є легким і швидким інструментом для написання і редагування коду. Завдяки простому інтерфейсу, підтримці багатьох мов програмування та мінімальному навантаженню на систему, Notepad++ популярний серед розробників, які шукають ефективний та легкий у використанні редактор.

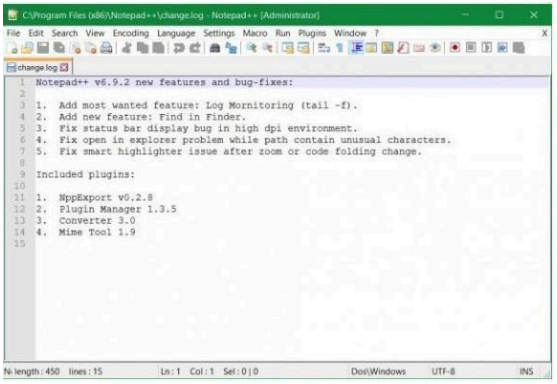

Рисунок 2.4. Інтерфейс користувача Notepad++

Розглянемо функціонал Notepad++:

Підсвічування синтаксису та складання коду - дозволяє легше читати код завдяки підсвічуванню ключових слів і можливості автоматичного форматування.

Підтримка макросів - забезпечує автоматизацію повторюваних дій шляхом запису макросів, що значно прискорює роботу.

Робота з вкладками - зручно відкривати і редагувати кілька файлів одночасно за допомогою вкладок, що полегшує багатозадачність.

Автозбереження та відновлення сесії - дозволяє автоматично зберігати зміни та відновлювати сесію після аварійного завершення роботи редактора.

Розглянемо переваги, що були виявлені користувачами при використанні Notepad++:

1. Легкість і швидкість роботи - Notepad++ працює швидко навіть на слабких комп'ютерах завдяки низьким вимогам до ресурсів.

2. Повністю безкоштовний - доступний для завантаження і використання безкоштовно, що робить його зручним вибором для будь-якого користувача.

Розглянемо недоліки, що були виявлені користувачами при використанні Notepad++:

1. Обмежена функціональність порівняно з іншими редакторами - немає вбудованих інструментів для дебагінгу або терміналу, що робить його менш функціональним для складних проєктів.
2. Тільки для Windows - Notepad++ офіційно доступний лише для Windows, що обмежує його використання на інших операційних системах.

Notepad++ - це надійний і легкий редактор, ідеальний для швидких змін та невеликих проєктів, особливо для тих, хто цінує мінімальне навантаження на систему. Проте, через обмежену функціональність він може не відповідати потребам складніших проєктів або професійної розробки на різних платформах [13]. Проаналізувавши всі переваги та недоліки наведених вище інструментів для реалізації проектованого програмного продукту було обрано SublimeText.

2.4 Аналіз та вибір ПЗ для тестування запитів до проектованого програмного забезпечення

Для тестування взаємодії користувача з нашим проектованим програмним забезпеченням у мережі нам знадобиться ПЗО, яке надасть нам змогу ініціювати запит до розробленого нам програмного забезпечення з передачею інформації для взаємодії. API клієнти - це інструменти середнього рівня складності. Це як правило десктопні програми, які дозволяють звернутися до ендпойнтів. Також вони надають додаткові можливості використання змінних, рандомайзерів, скриптів та інших функцій [14]. Розглянемо найбільш актуальні та сучасні інструменти , які задовольняють цим вимогам:

Insomnia (Рисунок 2.5)[15] - це популярний інструмент для тестування API, який забезпечує розробникам зручний інтерфейс для роботи з REST та

GraphQL запитами. Цей інструмент дозволяє швидко налаштовувати запити, переглядати та аналізувати відповіді сервера, працювати з аутентифікацією та додавати різноманітні параметри для глибшого тестування.

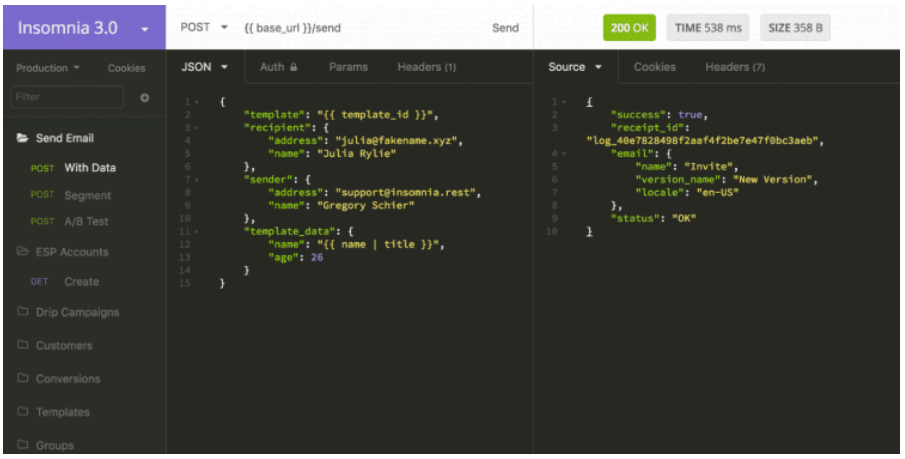

Рисунок 2.5. Інтерфейс користувача Insomnia

Insomnia також підтримує роботу з колекціями запитів, що робить процес тестування структурованим та організованим. Він сумісний з різними форматами передачі даних, включаючи JSON, XML, та Form URL-encoded, що робить його універсальним у використанні.

Розглянемо переваги, що були виявлені користувачами при використанні Insomnia:

1. Інтуїтивний інтерфейс - зрозумілий і простий у використанні, що дозволяє швидко створювати запити навіть новачкам.
2. Зручна робота з токенами аутентифікації - Insomnia спрощує додавання токенів, що необхідні для роботи з захищеними API.
3. Налаштування середовищ (environment) - дозволяє зберігати змінні для різних середовищ (наприклад, продакшн, тестування), що дуже зручно для управління запитами.

Розглянемо недоліки, що були виявлені користувачами при використанні Insomnia:

1. Відсутність багатьох інтеграцій - Insomnia має менше інтеграцій з іншими інструментами порівняно з альтернативами, такими як Postman.
2. Обмежена кастомізація - інструмент не дозволяє вносити значні зміни в інтерфейс або функціонал, що може обмежувати професійних користувачів.
3. Високе споживання ресурсів на великих проєктах - при роботі з великими колекціями запитів споживання пам’яті може значно зрости.

Insomnia - це простий у використанні та ефективний інструмент для тестування API, що відмінно підходить для роботи з REST і GraphQL. Його можливості роблять його чудовим вибором для розробників, які шукають простоту та швидкість у створенні та перевірці API-запитів, хоча деякі обмеження можуть стати перепоною для більш складних сценаріїв тестування або інтеграцій.

Postman (Рисунок 2.6)[18] - це потужний інструмент для розробників, призначений для тестування API. Він підтримує створення, налагодження та автоматизацію REST, SOAP та GraphQL запитів. Postman дозволяє зручно налаштовувати запити, переглядати відповіді серверів, а також забезпечує підтримку колекцій, що полегшує організацію запитів для великих проєктів. Окрім базового функціоналу, Postman пропонує можливості для налаштування середовищ, створення автоматизованих тестів, інтеграцію з CI/CD системами та підтримує спільну роботу над проєктами в команді, що робить його важливим інструментом для API-розробки.

Розглянемо переваги, що були виявлені користувачами при використанні Postman:

1. Розширені функції тестування та автоматизації - Postman підтримує написання тестів для запитів, що дозволяє легко автоматизувати перевірку API.

2. Спільна робота в команді - зручний інтерфейс для спільної роботи над запитами та проєктами, що дозволяє команді синхронізувати зміни та працювати над API в реальному часі.
3. Інтеграція з CI/CD інструментами - Postman легко інтегрується з популярними системами CI/CD, що дозволяє включати тестування API у загальний процес розробки.

Розглянемо недоліки, що були виявлені користувачами при використанні Postman:

1. Високе споживання ресурсів - на великих проєктах Postman може значно навантажувати систему, що впливає на швидкість роботи.
2. Платні функції для професійного використання - деякі важливі функції, як-от розширені можливості спільної роботи, доступні лише у платних версіях.
3. Складність інтерфейсу для новачків - через багатий функціонал новим користувачам може знадобитися час на освоєння інтерфейсу та всіх доступних функцій.

Загалом, Postman є потужним інструментом для тестування API, який підходить як для індивідуального використання, так і для роботи в команді. Завдяки широкому функціоналу, він ідеальний для розробників, яким потрібні автоматизація та організація запитів, хоча високе споживання ресурсів та складність для новачків можуть бути недоліками при використанні Postman для простих або невеликих проєктів.

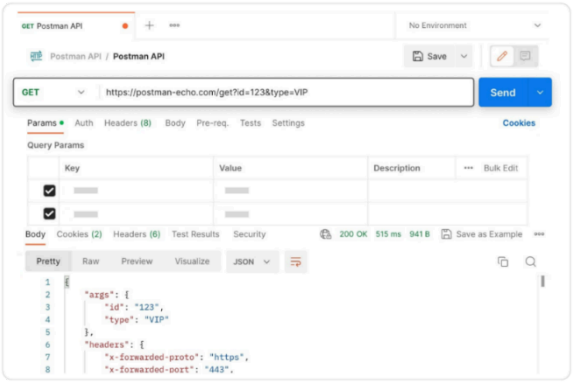

Рисунок 2.6. Інтерфейс користувача Postman

SOAPUI (Рисунок 2.7)[18] - це професійний інструмент для тестування веб-служб, спеціалізований на роботу з протоколами SOAP та REST.

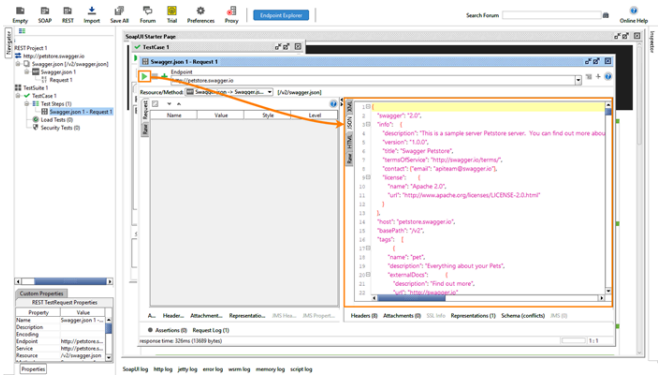

Рисунок 2.7. Інтерфейс користувача SOAPUI

Відомий своєю гнучкістю, SoapUI дозволяє створювати та виконувати запити, автоматизовані тести, перевірки безпеки та навантажувальні тести, що робить його особливо корисним для великих проєктів і комплексного тестування API. Цей інструмент підтримує функції побудови сценаріїв, налаштування середовищ, перевірки відповідей та параметризацію тестів, що допомагає у створенні гнучких і складних тестів для веб-служб.

Розглянемо переваги, що були виявлені користувачами при використанні SoapUI:

1. Розширені можливості для тестування SOAP - інструмент надає глибоку підтримку SOAP-запитів, що дозволяє тестувати всі аспекти веб-служб, заснованих на SOAP.
2. Автоматизація та налаштування сценаріїв - SoapUI дозволяє створювати складні сценарії для автоматизованого тестування, що полегшує роботу з великими API-проєктами.
3. Можливості навантажувального тестування - інструмент забезпечує можливість виконання навантажувальних тестів для оцінки продуктивності API під великим навантаженням.

Розглянемо недоліки, що були виявлені користувачами при використанні SoapUI:

1. Високе споживання ресурсів - під час виконання складних або навантажувальних тестів SoapUI може значно навантажувати систему.
2. Обмежені можливості інтерфейсу для REST API - хоча SoapUI підтримує REST, він не такий зручний для цього протоколу порівняно з іншими інструментами.
3. Платна версія для розширеного функціоналу - розширені можливості, такі як навантажувальні тести і детальні налаштування безпеки, доступні лише у платній версії.

У цілому, SoapUI є ефективним інструментом для тестування веб-служб, особливо тих, що побудовані на SOAP. Це чудовий вибір для проєктів, де потрібна автоматизація складних сценаріїв та оцінка продуктивності API, але його високе споживання ресурсів і обмежена зручність для REST можуть зробити його менш привабливим для деяких команд, особливо тих, що працюють лише з REST API.

Обираючи інструмент для тестування запитів до програмного забезпечення, що реалізує інтерактивний програмний додаток управління цифровими зображеннями, було враховано низку важливих аспектів з метою забезпечення ефективності, надійності та зручності процесу тестування. На основі проведеного аналізу було визначено, що Postman є оптимальним інструментом для поставлених потреб, з огляду на такі переваги:

1. Підтримка різних типів API - хоча додаток може бути орієнтований на локальну роботу з даними, окремі функції (наприклад, синхронізація, резервне копіювання або інтеграція з хмарними сервісами) можуть потребувати взаємодії через мережеві запити. Postman дозволяє тестувати HTTP-запити різних типів, що дає змогу перевірити коректність обміну даними у разі подальшого розширення функціоналу.

2. Зручний та інтуїтивно зрозумілий інтерфейс - у Postman легко створювати, надсилати та аналізувати запити і відповіді. Це спрощує тестування окремих модулів системи, зокрема операцій завантаження, отримання списку

зображень, оновлення метаданих, фільтрації/сортування, а також перевірки доступу до ресурсів.

3. Можливості автоматизації - Postman підтримує створення колекцій запитів, їх групове виконання та написання автоматизованих перевірок. Це дозволяє оперативно контролювати стабільність роботи компонентів під час внесення змін, а також зменшує ймовірність помилок при повторному тестуванні.

4. Активна спільнота та широка документація - Postman має значну базу знань, приклади та матеріали для вирішення типових задач. Це пришвидшує налаштування середовища тестування та підвищує загальну ефективність процесу розробки і перевірки програмного продукту [15].

5. Таким чином, використання Postman забезпечує гнучке й масштабоване тестування функціональних модулів та можливих серверних компонентів, що є важливою складовою якісної розробки інтерактивного програмного додатку управління цифровими зображеннями.

2.5 Аналіз та вибір інструментарію обраної мови програмування для реалізації функціоналу

Вибір інструментів для реалізації інтерактивного програмного додатку управління цифровими зображеннями є наступним із ключових етапів розробки. Такий програмний додаток повинен забезпечувати швидку обробку та відображення графічних даних, стабільну роботу інтерфейсу користувача, зручну навігацію між зображеннями, а також можливість локального збереження цифрового контенту та пов’язаних з ним метаданих. Тому важливо обрати технологічний стек, який дозволить ефективно реалізувати зазначений функціонал. Під час вибору інструментів враховувалися такі критерії: сумісність із мовою програмування, продуктивність, простота розробки, масштабованість, підтримка різних платформ і активність спільноти розробників.

Для створення інтерфейсу користувача було обрано React Native - кросплатформенний фреймворк, що дозволяє розробляти програмні додатки з єдиною кодовою базою для операційних систем Android та iOS. Даний фреймворк забезпечує підтримку адаптивного дизайну, плавну взаємодію з елементами інтерфейсу та можливість створення інтуїтивно зрозумілої навігації. Додатково використання платформи Expo значно спрощує процес тестування, налагодження, пакування та розгортання програмного додатку, що є важливим на етапах активної розробки. React Native має розвинену екосистему готових компонентів, численні бібліотеки та активну підтримку спільноти, що сприяє швидкій реалізації необхідного функціоналу.

Оскільки програмний додаток орієнтований на роботу з цифровими зображеннями без обов’язкового підключення до мережі Інтернет, пріоритет було надано локальній обробці та збереженню даних. Зображення та пов’язана з ними інформація зберігатимуться без використання серверних рішень чи зовнішніх сервісів. Для реалізації локального сховища даних було обрано бібліотеку @react-native-async-storage/async-storage, яка є стандартним інструментом для збереження інформації у форматі ключ–значення в середовищі React Native. Дана бібліотека забезпечує стабільну роботу на платформах Android та iOS, просту інтеграцію та можливість виконання базових операцій запису, читання, оновлення та видалення даних без складних налаштувань.

Для управління логікою програмного додатку та станом інтерфейсу було використано вбудовані можливості бібліотеки React, зокрема hooks (useState, useEffect та інші). Застосування даного підходу дозволяє ефективно організувати роботу з даними, забезпечити динамічне оновлення інтерфейсу та передбачувану поведінку додатку під час взаємодії користувача з цифровими зображеннями. Оскільки розроблюваний програмний додаток не передбачає складних сценаріїв керування станом або великої кількості взаємопов’язаних екранів, використання додаткових бібліотек для управління станом не є доцільним, що спрощує архітектуру та супровід програмного продукту.

Таким чином, обраний набір інструментів - React Native, Expo, AsyncStorage та hooks бібліотеки React - забезпечує простоту реалізації, стабільність роботи та підтримку офлайн-функціоналу, що є ключовим для інтерактивного програмного додатку управління цифровими зображеннями.

Висновки до розділу 2

У другому розділі було проведено аналіз сучасних технологій та інструментів програмної розробки, актуальних для реалізації інтерактивного програмного додатку управління цифровими зображеннями. Розглянуто основні характеристики мов програмування та виконано їх порівняння за рядом критеріїв, зокрема за парадигмами програмування, типізацією, особливостями керування пам’яттю, а також функціональними й об’єктно-орієнтованими можливостями. На основі отриманих результатів обґрунтовано вибір мови програмування JavaScript та кросплатформенного фреймворку React Native як основи для розробки програмного продукту.

Окремо було проаналізовано допоміжні засоби, необхідні для практичної реалізації та супроводу проєкту: середовище написання коду та інструменти тестування взаємодії із програмними компонентами. За результатами порівняння текстових редакторів було обрано Sublime Text, що забезпечує високу швидкодію та зручність роботи з кодом. Для тестування запитів і перевірки взаємодії із можливими API-компонентами було обрано Postman, який надає зручні засоби формування, надсилання та аналізу запитів.

Таким чином, сформовано узгоджений набір програмних засобів та інструментарію, який забезпечує кросплатформенність, стабільність роботи, зручність розробки й можливість подальшого розширення функціоналу додатку.

РОЗДІЛ 3

ПРОЕКТУВАННЯ ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ ІНТЕРАКТИВНОГО ПРОГРАМНОГО ДОДАТКУ УПРАВЛІННЯ ЦИФРОВИМИ ЗОБРАЖЕННЯМИ

3.1 Визначення функціональності та основних можливостей додатку

Першим етапом практичної реалізації поставленого завдання є визначення функціональності та основних можливостей проектованого інтерактивного програмного додатку управління цифровими зображеннями. На основі аналізу предметної області, виконаного у першому розділі роботи, було сформовано перелік функціональних вимог, які мають бути реалізовані в межах розробки мобільного застосунку. Дані вимоги відповідають сучасним потребам користувачів у зручному зберіганні, перегляді, впорядкуванні та обміні цифровими зображеннями за допомогою мобільних пристроїв.

Проєктований застосунок орієнтований на індивідуальних користувачів, які потребують інструменту для керування власною колекцією фотографій, організації зображень за альбомами та тегами, а також взаємодії з іншими користувачами шляхом перегляду, коментування та оцінювання зображень. До основних функціональних можливостей, які має реалізовувати даний програмний додаток, належать:

1. Реєстрація користувачів - необхідно забезпечити можливість створення нового облікового запису шляхом введення персональних даних користувача, зокрема електронної пошти та пароля, з подальшим збереженням даних у базі даних.

2. Авторизація користувачів - необхідно реалізувати механізм входу до системи за допомогою перевірки коректності введених облікових даних, що дозволить забезпечити персоналізований доступ до функціоналу застосунку.

3. Головна сторінка зображень - необхідно реалізувати головний екран, на якому відображатиметься список цифрових зображень із короткою

інформацією, зокрема назвою, попереднім переглядом, кількістю переглядів та тегами.

4. Пошук і фільтрація зображень - застосунок повинен надавати можливість здійснювати пошук зображень за назвою та описом, а також фільтрацію за тегами для спрощення навігації у великій кількості фотографій.

5. Детальний перегляд зображення - при виборі окремого зображення користувач повинен мати змогу переглянути його у повному розмірі разом із детальною інформацією, зокрема описом, тегами, кількістю переглядів, коментарями та середньою оцінкою.

6. Завантаження та додавання зображень - необхідно реалізувати можливість додавання нових зображень з галереї мобільного пристрою з подальшим введенням описових даних, таких як назва, опис та теги.

7. Керування альбомами - застосунок має забезпечувати створення, редагування та видалення альбомів, а також додавання до них зображень з метою логічної організації фотоколекції користувача.

8. Коментування та оцінювання зображень - необхідно надати користувачам можливість залишати коментарі до зображень та оцінювати їх за п’ятибальною шкалою, що сприяє підвищенню інтерактивності застосунку.

9. Керування рівнем доступу до зображень - застосунок повинен підтримувати налаштування приватності зображень (публічні, приватні або доступні за посиланням), що дозволяє користувачам контролювати доступ до власного контенту.

10. Адаптивний інтерфейс для мобільних пристроїв - інтерфейс застосунку має бути оптимізований для різних розмірів екранів мобільних пристроїв, забезпечуючи зручну навігацію, читабельність інформації та комфортну взаємодію з елементами керування.

Таким чином, визначені функціональні можливості охоплюють основні аспекти управління цифровими зображеннями у мобільному середовищі. Реалізація зазначеного функціоналу дозволяє створити повноцінний інтерактивний програмний додаток, який забезпечує ефективне зберігання,

впорядкування та використання цифрових фотографій відповідно до потреб користувачів.

3.2 Проєктування структури бази даних

Для забезпечення стабільного функціонування інтерактивного програмного додатку управління цифровими зображеннями необхідним етапом є проєктування структури бази даних. Оскільки робота системи передбачає зберігання інформації про користувачів, цифрові зображення, їхні властивості, а також пов’язані з ними альбоми, теги, коментарі та оцінки, виникає потреба у створенні продуманої та цілісної моделі зберігання даних. Коректне проєктування бази даних забезпечує цілісність, узгодженість та надійність інформації, а також створює умови для ефективної взаємодії між окремими компонентами системи.

З огляду на функціональні вимоги системи, перейдемо до формування структури реляційної бази даних, яка міститиме таблиці для зберігання користувачів, цифрових зображень, альбомів та допоміжних сутностей, а також забезпечуватиме між ними відповідні реляційні зв’язки.

3.2.1 Проєктування структури реляційної бази даних

Мобільний застосунок управління цифровими зображеннями оперує різними типами даних, зокрема обліковими записами користувачів, інформацією про завантажені зображення, їхні метадані, альбоми, теги, коментарі та оцінки. Для ефективного керування цими даними у межах розроблюваної системи передбачено використання реляційної бази даних, яка дозволяє встановлювати логічні зв’язки між окремими сутностями, забезпечувати цілісність інформації, а також виконувати складні вибірки, фільтрацію та сортування даних. Застосування реляційної моделі зберігання даних забезпечує структуроване представлення інформації, зручність її обробки та коректну роботу всіх функціональних модулів застосунку. Серверна частина системи використовує централізовану базу даних, яка взаємодіє з клієнтською

частиною через REST API та відповідає за обробку запитів, пов’язаних із завантаженням, переглядом, редагуванням і організацією цифрових зображень.

Перейдемо до проєктування основних таблиць, що формують ядро структури бази даних у розроблюваній системі. Першим елементом є таблиця користувачів, яка відповідає за зберігання облікових даних, профільної інформації та забезпечує авторизацію у системі. Структура таблиці користувачів наведена у таблиці 3.1.

Таблиця 3.1

Структура таблиці користувачів у проєктованій системі

Об’єкт	**Властивість**	**Тип**	**Розмірність**	**Ідентифікатор**
Таблиця користувачів	Ідентифікатор	INTEGER	100	id
	Ім’я	TEXT	255	name
	Телефон	TEXT	255	phone
	Вік	INTEGER	100	age
	Ел. пошта	TEXT	255	email
	Пароль	TEXT	255	password
	Аватар	TEXT	255	avatar_url
	Біографія	TEXT	255	bio
	Дата створення	DATETIME	–	created_at
	Дата оновлення	DATETIME	–	updated_at

У таблиці користувачів зберігається ключова інформація для автентифікації та ідентифікації. Поле id є первинним ключем із встановленим автоінкрементом. Поле email є унікальним і використовується як логін у системі. Поля name, phone, age містять персональні дані, тоді як password забезпечує авторизацію. Додатково передбачено поля avatar_url та bio, які формують розширену інформацію профілю користувача. Поля created_at і updated_at використовуються для контролю часу створення та зміни профілю.

Наступним кроком перейдемо до проєктування таблиці зображень, яка є центральною сутністю системи, оскільки саме зображення виступають основним об’єктом зберігання й взаємодії. У таблиці фіксуються як описові поля (назва, опис), так і технічні метадані файлу, параметри доступу та статистика переглядів. Структура таблиці зображень наведена у таблиці 3.2.

Таблиця 3.2

Структура таблиці зображень у проєктованій системі

Об’єкт	Властивість	Тип	Розмірність	Ідентифікатор
Таблиця зображень	Ідентифікатор	INTEGER	100	id
	Ел.пошта	TEXT	255	user_email
	Назва	TEXT	255	title
	Опис	TEXT	255	description
	Оригінальна назва файлу	TEXT	255	original_filename
	MIME-тип	TEXT	255	mime_type
	Розмір (байти)	INTEGER	100	size_bytes
	Ширина	INTEGER	100	width
	Висота	INTEGER	100	height
	Шлях до файлу	TEXT	255	storage_path
	Шлях до мініатюри	TEXT	255	thumbnail_path
	Видимість	TEXT	255	visibility
	Токен доступу	TEXT	255	share_token
	Перегляди	INTEGER	100	views_count
	Ознака видалення	INTEGER	100	is_deleted
	Видалення	DATETIME	–	deleted_at
	Дата створення	DATETIME	–	created_at
	Дата оновлення	DATETIME	–	updated_at

Таблиця зображень містить інформацію про всі завантажені у систему файли. Поле user_email формує зовнішній ключ і зв’язує зображення з конкретним користувачем (власником). Поля title і description використовуються для зручного пошуку та відображення у користувацькому інтерфейсі. Технічні властивості файлу зберігаються у полях original_filename, mime_type, size_bytes, width, height. Поле storage_path містить шлях до основного зображення, а thumbnail_path може містити шлях до зменшеної копії для відображення у списках. Поле visibility визначає рівень доступу до зображення (public/private/link), а поле share_token використовується для реалізації доступу за посиланням. Поле views_count застосовується для збереження статистики переглядів. Для реалізації механізму м’якого видалення використовується пара полів is_deleted та deleted_at, що дозволяє приховати зображення без фізичного видалення запису з бази даних.

Наступним кроком перейдемо до проєктування бази тегування, яка потрібна для класифікації зображень і реалізації фільтрації. Для цього використовуються таблиця тегів та проміжна таблиця зв’язку «багато-до-багатьох». Структура таблиці тегів наведена у таблиці 3.3.

Таблиця 3.3

Структура таблиці тегів у проєктованій системі

Об’єкт	Властивість	Тип	Розмірність	Ідентифікатор
Таблиця тегів	Ідентифікатор	INTEGER	100	id
	Назва тегу	TEXT	255	name
	Дата створення	DATETIME	–	created_at

У таблиці тегів зберігаються унікальні назви тегів. Поле name має обмеження унікальності, що запобігає дублюванню однакових тегів у системі. Оскільки одне зображення може мати декілька тегів, а один тег може належати багатьом зображенням, реалізуємо таблицю зв’язку image_tags. Її структуру наведено у таблиці 3.4.

Таблиця 3.4

Структура таблиці зв’язку image_tags у проєктованій системі

Об’єкт	Властивість	Тип	Розмірність	Ідентифікатор
Таблиця image_tags	Ідентифікатор зображення	INTEGER	100	image_id
	Ідентифікатор тегу	INTEGER	100	tag_id

Таблиця image_tags реалізує зв’язок «багато-до-багатьох». Первинним ключем є складений ключ (image_id, tag_id), що гарантує відсутність дублюючих прив’язок. Для обох полів визначено зовнішні ключі із каскадним видаленням, що забезпечує цілісність зв’язків.

Наступним кроком перейдемо до проєктування структури таблиці альбомів, яка призначена для організації цифрових зображень користувача у впорядковані логічні колекції. Альбоми дозволяють групувати зображення за тематикою, подіями або будь-якими іншими критеріями, що значно спрощує навігацію та управління великим обсягом медіаконтенту. Кожен альбом має власні атрибути, зокрема назву та опис, які використовуються для його

ідентифікації у застосунку, а також параметри видимості, що визначають рівень доступу до вмісту альбому (публічний, приватний або доступний за посиланням).

Додатково передбачено використання унікального токена доступу, який дозволяє реалізувати механізм поширення альбому за посиланням без необхідності авторизації інших користувачів у системі. Зв’язок альбому з конкретним користувачем забезпечується шляхом збереження електронної пошти власника, що дає змогу відображати та керувати лише власними альбомами. Структура таблиці альбомів наведена у таблиці 3.5.

Таблиця 3.5

Структура таблиці альбомів у проєктованій системі

Об’єкт	**Властивість**	**Тип**	**Розмірність**	**Ідентифікатор**
Таблиця альбомів	Ідентифікатор	INTEGER	100	id
	Електронна пошта користувача	TEXT	255	user_email
	Назва	TEXT	255	name
	Опис	TEXT	255	description
	Видимість	TEXT	255	visibility
	Токен доступу	TEXT	255	share_token
	Дата створення	DATETIME	–	created_at
	Дата оновлення	DATETIME	–	updated_at

У таблиці альбомів зберігається інформація про логічні колекції зображень, які створюються користувачами для впорядкування власного медіаконтенту. Поле user_email встановлює зв’язок між альбомом та конкретним користувачем, що дозволяє забезпечити розмежування доступу та коректне відображення альбомів у межах профілю. Поля name та description містять текстову інформацію про альбом і використовуються для ідентифікації та зручності навігації. Атрибут visibility визначає рівень доступу до альбому (публічний, приватний або доступний за посиланням), а поле share_token застосовується для реалізації функції поширення альбому за унікальним URL. Поля created_at та updated_at забезпечують контроль змін і дозволяють відстежувати історію створення та редагування альбомів.

Наступним кроком перейдемо до проєктування структури таблиці зв’язку між альбомами та зображеннями. Оскільки один альбом може містити декілька зображень, а одне зображення може одночасно входити до різних альбомів, у системі використовується зв’язок типу «багато-до-багатьох». Для реалізації такого зв’язку передбачено використання проміжної таблиці album_images, структура якої наведена у таблиці 3.6.

Таблиця 3.6

Структура таблиці зв’язку album_images у проєктованій системі

Об’єкт	**Властивість**	**Тип**	**Розмірність**	**Ідентифікатор**
Таблиця album_images	Ідентифікатор альбому	INTEGER	100	album_id
	Ідентифікатор зображення	INTEGER	100	image_id
	Позиція у альбомі	INTEGER	100	position
	Дата додавання	DATETIME	–	added_at

Таблиця album_images виконує роль зв’язувальної сутності між таблицями альбомів та зображень. Складений первинний ключ (album_id, image_id) гарантує, що одне й те саме зображення не може бути додане до одного альбому більше одного разу. Поле position використовується для збереження порядку відображення зображень всередині альбому, що дозволяє реалізувати сортування або ручне впорядкування. Атрибут added_at фіксує момент додавання зображення до альбому та може бути використаний для аналітики або відображення хронології змін.

Наступним кроком перейдемо до проєктування структури таблиці оцінювання зображень. Для реалізації функціоналу оцінювання зображень користувачами у застосунку передбачено окрему таблицю ratings, яка дозволяє зберігати індивідуальні оцінки та формувати середній рейтинг зображення. Структура таблиці оцінок наведена у таблиці 3.7.

Таблиця 3.7

Структура таблиці оцінок ratings у проєктованій системі

Об’єкт	Властивість	Тип	Розмірність	Ідентифікатор
Таблиця ratings	Ідентифікатор	INTEGER	100	id
	Ідентифікатор зображення	INTEGER	100	image_id
	Електронна пошта користувача	TEXT	255	user_email
	Значення оцінки	INTEGER	100	value
	Дата створення	DATETIME	–	created_at
	Дата оновлення	DATETIME	–	updated_at

У таблиці ratings зберігається інформація про оцінки, які користувачі виставляють зображенням за п’ятибальною шкалою. Поле value має обмеження значень (від 1 до 5), що гарантує коректність даних. Зв’язок із зображенням забезпечується через поле image_id, а з користувачем - через user_email. Обмеження унікальності пари (image_id, user_email) запобігає можливості повторного оцінювання одно

го зображення тим самим користувачем. Поля created_at та updated_at дозволяють фіксувати час встановлення або зміни оцінки.

Наступним кроком перейдемо до проєктування структури таблиці коментарів. Для забезпечення можливості зворотного зв’язку та соціальної взаємодії між користувачами у застосунку передбачено функціонал коментування зображень. Такий підхід дозволяє користувачам обмінюватися враженнями, залишати зауваження та брати участь у обговоренні візуального контенту, що підвищує загальну інтерактивність системи. З метою реалізації зазначеного функціоналу використовується таблиця comments, структура якої наведена у таблиці 3.8, і яка забезпечує збереження тексту коментарів, зв’язок із відповідними зображеннями та користувачами, а також можливість подальшого розширення функціоналу.

Таблиця comments використовується для зберігання текстових коментарів, залишених користувачами під зображеннями. Поле content містить безпосередній текст повідомлення. Атрибути image_id та user_email формують

зв’язки з відповідним зображенням та автором коментаря. Поля created_at і updated_at забезпечують контроль часу створення та редагування коментарів, що дозволяє реалізувати функцію оновлення або модерації контенту.

Таблиця 3.8

Структура таблиці коментарів у проєктованій системі

Об’єкт	Властивість	Тип	Розмірність	Ідентифікатор
Таблиця comments	Ідентифікатор	INTEGER	100	id
	Ідентифікатор зображення	INTEGER	100	image_id
	Електронна пошта користувача	TEXT	255	user_email
	Текст коментаря	TEXT	255	content
	Дата створення	DATETIME	–	created_at
	Дата оновлення	DATETIME	–	updated_at

Таким чином, спроєктована структура реляційної бази даних охоплює всі ключові сутності предметної області та забезпечує коректні зв’язки між користувачами, зображеннями, альбомами, тегами, оцінками й коментарями. Використання зовнішніх ключів, складених первинних ключів і обмежень унікальності гарантує цілісність даних та узгоджену роботу системи.

3.2.2 Проєктування ER-діаграми бази даних

Після визначення складу основних таблиць бази даних та опису їх структури необхідно сформувати візуальне представлення логічних зв’язків між сутностями, що використовуються у проєктованій системі управління цифровими зображеннями. Для цього застосовується ER-діаграма (Entity-Relationship), яка дозволяє наочно відобразити сутності бази даних, їх атрибути та типи взаємозв’язків між ними. ER-діаграма є важливим етапом логічного проєктування, оскільки вона допомагає перевірити цілісність структури та відповідність бази даних функціональним вимогам застосунку.

ER-діаграма відображає взаємодію між основними сутностями системи: «Користувач», «Зображення», «Альбом», «Тег», «Коментар» та «Оцінка». Користувач може створювати зображення та альбоми, залишати коментарі й

оцінювати зображення інших користувачів. Для реалізації можливості присвоєння кількох тегів одному зображенню, а також додавання зображень до альбомів, використано проміжні таблиці, що формують зв’язки типу «багато-до-багатьох». Такий підхід забезпечує гнучкість структури та запобігає дублюванню даних.

На рисунку 3.1 наведено ER-діаграму логічної структури бази даних мобільного застосунку управління цифровими зображеннями.

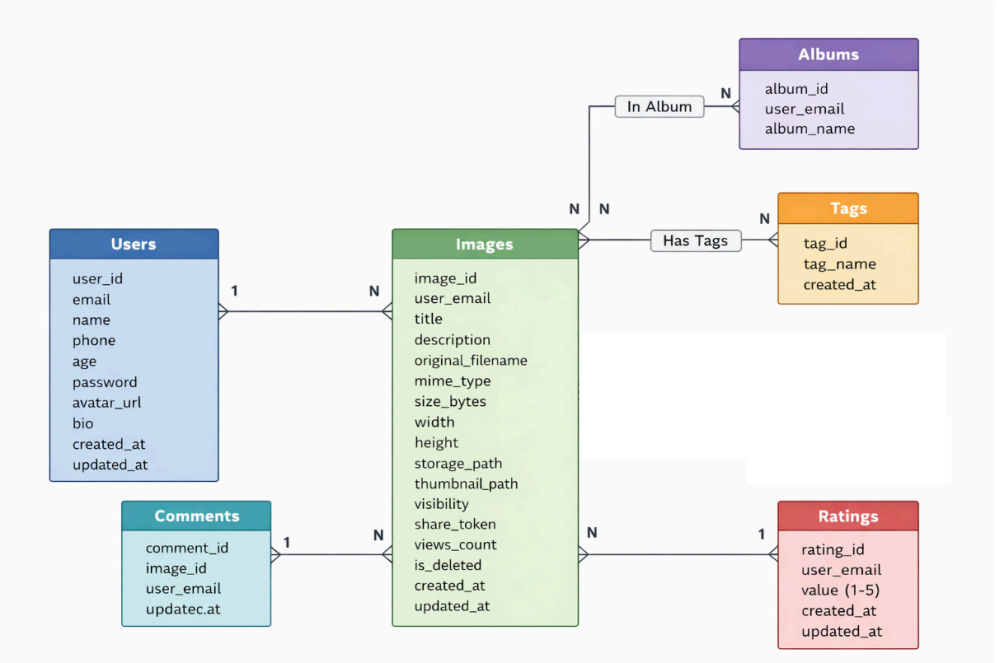

Рисунок 3.1. ER-діаграма логічної структури бази даних мобільного додатку управління цифровими зображеннями

Використання ER-діаграми дозволяє переконатися у коректності визначених зв’язків між сутностями та ефективності побудованої логічної моделі. На основі цієї діаграми реалізовано фізичну структуру бази даних із використанням SQLite у серверній частині застосунку. Таким чином, логічне проєктування бази даних завершено та створено основу для подальшої практичної реалізації системи.

3.3 Програмна реалізація проєктованого додатку

На основі визначеної архітектури системи та спроєктованої структури бази даних перейдемо до програмної реалізації мобільного застосунку управління цифровими зображеннями. Клієнтська частина додатку розробляється з використанням мови програмування JavaScript та фреймворку React Native, що забезпечує можливість створення кросплатформених мобільних застосунків для операційних систем Android та iOS. Обрана технологія дозволяє поєднати зручність розробки, високу продуктивність і доступ до нативних можливостей мобільних пристроїв.

Функціональні можливості застосунку будуть реалізовані у вигляді окремих логічних екранів, кожен з яких відповідатиме певному етапу взаємодії користувача із системою: реєстрації та авторизації, перегляду зображень, їх додавання, організації в альбоми та детального перегляду. Під час проєктування інтерфейсу особлива увага приділяється зручності навігації, логічному розміщенню елементів та швидкому доступу до основних функцій. Інтерфейс застосунку проєктується з урахуванням принципів UX/UI, що забезпечує інтуїтивну взаємодію з користувачем і наочне подання інформації. Користувач матиме можливість здійснювати пошук і фільтрацію зображень за тегами, змінювати параметри видимості, оцінювати та коментувати зображення, а також організовувати їх у тематичні альбоми. Такий підхід створює основу для ефективного управління цифровими зображеннями та подальшої реалізації всіх функціональних модулів системи.

3.3.1 Програмна реалізація інтерфейсу реєстрації користувача

Першим кроком реалізуємо механізм реєстрації користувача для забезпечення можливості створення індивідуального облікового запису в мобільному застосунку для керування цифровими зображеннями. Реєстрація є необхідною для персоналізації користувацького досвіду, збереження інформації щодо зображень та отримання доступу до статистики. Реєстрація передбачає

введення обов’язкових даних: імені, телефону, віку, електронної пошти та пароля. Після створення облікового запису користувач автоматично авторизується в системі. Почнемо з реалізації інтерфейсу екрану RegisterScreen, за допомогою якого користувач може ввести свої дані та ініціювати процес реєстрації:

```
<View style={styles.container}>
  <Text style={styles.title}>Реєстрація</Text>
  <TextInput style={styles.input} placeholder="Ім'я" value={name} onChangeText={setName} />
  <TextInput style={styles.input} placeholder="Email" value={email} onChangeText={setEmail} />
  <TextInput style={styles.input} placeholder="Пароль" value={password} onChangeText={setPassword}
secureTextEntry />
  <Button title="Зареєструватися" onPress={handleRegister} />
</View>
```

У наведеному фрагменті коду реалізовано інтерфейс екране реєстрації, що містить текстові поля для введення основних даних користувача та кнопку підтвердження. Усі введені значення зберігаються у відповідних змінних стану, які оновлюються за допомогою функцій setName, setPhone, setAge, setEmail та setPassword. Натискання кнопки "Зареєструватися" ініціює виклик функції handleRegister, яка здійснює перевірку та відправлення даних на сервер. Наступним кроком реалізуємо функцію handleRegister(), яка відповідає за обробку події реєстрації користувача:

```
const handleRegister = async () => {
  if (!name || !phone || !age || !email || !password)
    return Alert.alert('Помилка', 'Заповніть усі поля.');
  try {
    const res = await fetch('http://192.168.1.104:3001/api/register', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ name, phone, age: parseInt(age), email, password }),
    });
    const data = await res.json();
    if (res.status === 201) { await AsyncStorage.setItem('currentUser', email); signIn(email); }
    else Alert.alert('Помилка', data.message || 'Щось пішло не так');
  } catch { Alert.alert('Помилка', 'Проблема з сервером. Спробуйте пізніше.'); }
```

У наведеному фрагменті коду реалізовано функцію handleRegister(), яка виконує перевірку полів форми на наявність обов’язкових даних. У разі правильного заповнення формується POST-запит до серверного API /api/register,

у тілі якого передаються дані нового користувача. У разі успішної відповіді (код 201) система зберігає електронну пошту користувача в AsyncStorage та виконує вхід у систему за допомогою функції signIn(). У протилежному випадку виводиться повідомлення про помилку.

З боку сервера логіка обробки запиту реєстрації реалізована у вигляді маршруту /api/register, що забезпечує перевірку та збереження даних у базу SQLite:

```
app.post('/api/register', async (req, res) => {
  const { name, phone, age, email, password } = req.body;
  try {
    const db = await open({ filename: './db.sqlite', driver: sqlite3.Database });
    const existing = await db.get('SELECT * FROM users WHERE email = ?', [email]);
    if (existing) return res.status(400).json({ message: 'Користувач з таким email вже існує' });
    await db.run(
      'INSERT INTO users (name, phone, age, email, password) VALUES (?, ?, ?, ?, ?)',
      [name, phone, age, email, password]
    );
    res.status(201).json({ message: 'Користувача успішно зареєстровано' });
  } catch { res.status(500).json({ message: 'Внутрішня помилка сервера' }); }
});
```

У наведеному фрагменті коду реалізовано серверний маршрут обробки запиту реєстрації нового користувача. Система зчитує дані з тіла HTTP-запиту, здійснює перевірку на наявність користувача з такою ж електронною поштою, і в разі його відсутності - додає новий запис до бази даних SQLite. У випадку успішного завершення операції сервер надсилає відповідь із кодом 201. У разі помилки або конфлікту повертається відповідне повідомлення з поясненням. Результат реалізації інтерфейсу реєстрації користувача наведено на рисунку 3.2.

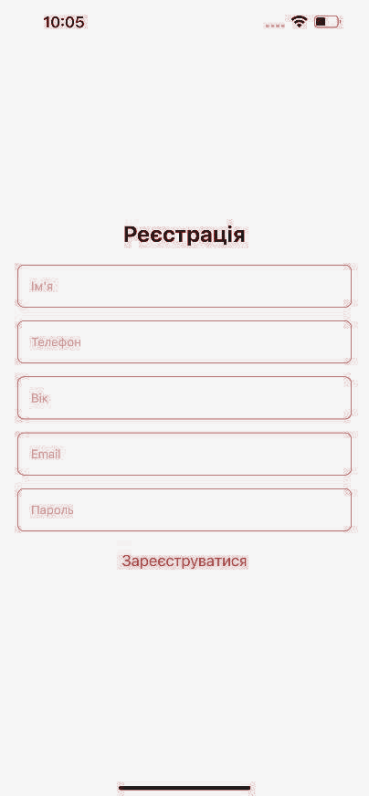

Рисунок 3.2. Реалізація інтерфейсу реєстрації користувача у додатку

Таким чином, на цьому етапі реалізовано повноцінний механізм реєстрації користувача із збереженням даних на сервері та подальшою авторизацією. Реєстраційна форма перевіряє обов’язкові поля, надсилає дані через HTTP-запит до API та забезпечує збереження активного сеансу через локальне сховище.

3.3.2 Програмна реалізація інтерфейсу авторизації користувача

Наступним кроком реалізуємо логіку авторизації користувача, яка забезпечує перевірку введених облікових даних та надає доступ до персоналізованих функцій мобільного застосунку для керування цифровими зображеннями. Авторизація необхідна для ідентифікації користувача, збереження даних його зображень, синхронізації доступу до них, відображення статистики роботи з додатком та коментарів. У процесі авторизації користувач вводить електронну пошту та пароль, які перевіряються шляхом надсилання запиту до серверної частини, де здійснюється відповідний запит до бази даних.

У разі успішної перевірки користувача авторизують у системі, після чого здійснюється перехід до головного інтерфейсу застосунку. Почнемо з реалізації функції обробки авторизації користувача у клієнтській частині застосунку:

```
const handleLogin = async () => {
  if (!email || !password) return Alert.alert('Помилка', 'Введіть email і пароль.');
  try {
    const res = await fetch('http://192.168.1.104:3001/api/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ email, password }),
    });
    const data = await res.json();
    if (res.ok) { await AsyncStorage.setItem('currentUser', email); signIn(email); }
    else Alert.alert('Помилка', data.message || 'Невірний email або пароль');
  } catch { Alert.alert('Помилка', 'Проблема з підключенням до сервера'); }
};
```

У наведеній частині коду реалізовано функцію handleLogin(), яка виконує авторизацію користувача. Спочатку система перевіряє, чи заповнені поля електронної пошти та пароля. Далі за допомогою fetch відправляється POST-запит до серверного маршруту /api/login, де передаються введені дані у форматі JSON. Якщо сервер повертає позитивну відповідь, застосунок зберігає адресу електронної пошти користувача у локальному сховищі (AsyncStorage) і викликає функцію signIn, що оновлює контекст автентифікації. У разі помилки відображається відповідне повідомлення. Наступним кроком реалізуємо інтерфейс екрана авторизації користувача:

```
<View style={styles.container}>
  <Text style={styles.title}>Вхід</Text>
  <TextInput style={styles.input} placeholder="Email" value={email} onChangeText={setEmail}
keyboardType="email-address" />
  <TextInput style={styles.input} placeholder="Пароль" value={password} onChangeText={setPassword}
secureTextEntry />
  <Button title="Увійти" onPress={handleLogin} />
  <TouchableOpacity onPress={() => navigation.navigate('Register')}>
    <Text style={styles.registerText}>Зареєструватися</Text>
  </TouchableOpacity>
</View>
```

У наведеній частині коду реалізовано інтерфейс екрана авторизації. Він містить два текстові поля для введення електронної пошти та пароля, кнопку

для входу до системи та елемент TouchableOpacity, який перенаправляє користувача на екран реєстрації у разі відсутності акаунта. Компонування елементів забезпечується за допомогою контейнера View, а стилі визначені окремо у стилізаційному об’єкті StyleSheet. Використання керованих компонентів дозволяє відстежувати введені користувачем дані у режимі реального часу. Такий підхід забезпечує зручну взаємодію з інтерфейсом та підвищує передбачуваність поведінки екрана авторизації.

На серверному боці реалізуємо обробку запиту авторизації користувача за допомогою маршруту /api/login:

```
app.post('/api/login', (req, res) => {
  const { email, password } = req.body;
  db.get(
    'SELECT * FROM users WHERE email = ? AND password = ?',
    [email, password],
    (err, user) => {
      if (err) return res.status(500).json({ message: 'Помилка на сервері' });
      if (!user) return res.status(401).json({ message: 'Невірний email або пароль' });
      res.json({ message: 'Вхід успішний', user });
    }
  );
});
```

У наведеній частині коду реалізовано обробку запиту на авторизацію користувача в Node.js з використанням бази даних SQLite. Після отримання POST-запиту, що містить електронну пошту та пароль, сервер формує SQL-запит для перевірки наявності відповідного користувача в таблиці users. У разі успішного знаходження запису система повертає JSON-відповідь із повідомленням про успішний вхід та об’єктом користувача. Якщо відповідного запису не знайдено, клієнт отримує повідомлення про невірні дані для входу. Також реалізовано обробку можливих помилок при взаємодії з базою даних - наприклад, у разі збоїв підключення або порушення синтаксису SQL-запиту сервер надсилає клієнту повідомлення про внутрішню помилку. Такий підхід забезпечує базовий рівень безпеки та стабільності під час авторизації у системі.

Результат реалізації інтерфейсу авторизації користувача наведено на рисунку 3.3.

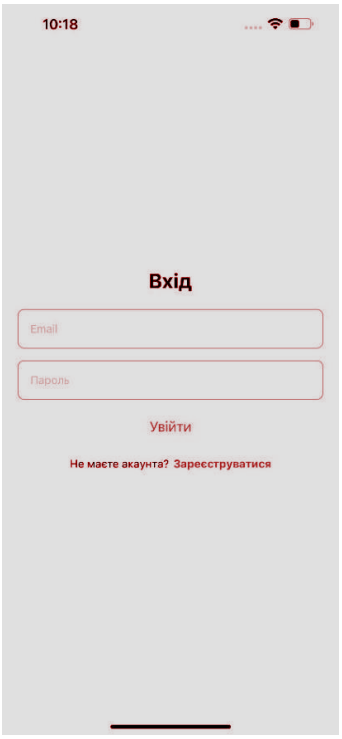

Рисунок 3.3. Інтерфейс авторизації користувача у реалізованому додатку

Таким чином реалізовано повноцінну логіку авторизації, що забезпечує перевірку даних на стороні клієнта та сервера, обробку помилок і безпечний доступ користувача до персоналізованого функціоналу застосунку.

3.3.3 Програмна реалізація головної сторінки мобільного застосунку

Наступним кроком перейдемо до реалізації головної сторінки мобільного застосунку управління цифровими зображеннями, яка є ключовим елементом інтерфейсу користувача. На цій сторінці відображається динамічний список цифрових зображень, що зберігаються у серверній базі даних і пов’язані з поточним авторизованим користувачем або доступні для публічного перегляду. Основною метою цього етапу є забезпечення можливості користувачу здійснювати перегляд, пошук та відкриття зображень, а також перехід до екрану додавання нового зображення. Функціонал головної сторінки має бути зручним, інформативним та адаптованим для мобільних пристроїв. Реалізація включає формування запиту до серверної частини для отримання списку зображень,

збереження отриманих даних у стані компонента, а також обробку пошуку та фільтрації.

Перейдемо до поетапної реалізації логіки даного функціоналу.

```
const MainPage = () => {
  const navigation = useNavigation();
  const [images, setImages] = useState([]);
  const [filteredImages, setFilteredImages] = useState([]);
  const [search, setSearch] = useState('');
  const [selectedTag, setSelectedTag] = useState('all');
  const [showOnlyMine, setShowOnlyMine] = useState(true);
  const [loading, setLoading] = useState(true);
  const [userEmail, setUserEmail] = useState(null);
};
```

У наведеній частині коду реалізовано створення функціонального компонента MainPage, який відповідає за логіку головної сторінки застосунку. За допомогою хуків useState ініціалізуються змінні стану: список усіх зображень (images), відфільтрований список (filteredImages), текст пошукового запиту (search), обраний тег (selectedTag), режим відображення (власні або публічні зображення), індикатор завантаження (loading) та електронна пошта поточного користувача (userEmail). Хук useNavigation використовується для навігації між екранами застосунку.

Наступним кроком перейдемо до реалізації механізму завантаження зображень із серверної частини та їх збереження у стані компонента.

```
const loadImages = useCallback(async () => {
  try {
    setLoading(true);
    const token = await AsyncStorage.getItem('userToken');
    if (!token) { setUserEmail(null); setImages([]); setFilteredImages([]); return; }
    setUserEmail(token);
    const url = showOnlyMine
      ? `${BASE_URL}/api/images?userEmail=${encodeURIComponent(token)}`
      : `${BASE_URL}/api/images?publicOnly=1`;
    const data = await (await fetch(url)).json();
    setImages(data); setFilteredImages(data);
  } catch (err) { console.error('Помилка завантаження зображень:', err); }
  finally { setLoading(false); }
}, [showOnlyMine]);
```

У наведеній частині коду реалізовано функцію loadImages, яка виконує асинхронне отримання зображень із серверної частини системи. Перед

виконанням запиту зчитується електронна пошта користувача з локального сховища AsyncStorage, що дозволяє ідентифікувати активну сесію. Залежно від обраного режиму формується відповідний HTTP-запит - для отримання власних або публічних зображень. Отримані дані зберігаються у станах images та filteredImages, що дозволяє надалі виконувати фільтрацію без повторних запитів до сервера. Хук useFocusEffect забезпечує автоматичне оновлення даних при кожному поверненні на головну сторінку.

Наступним кроком перейдемо до реалізації логіки пошуку та фільтрації зображень.

```
const handleSearchChange = (text) => {
  setSearch(text);
  const q = text.toLowerCase();
  setFilteredImages(!q ? images : images.filter(img =>
    img.title.toLowerCase().includes(q) ||
    (img.description && img.description.toLowerCase().includes(q))
  ));
};

const handleTagChange = (tag) => {
  setSelectedTag(tag);
  setFilteredImages(tag === 'all' ? images : images.filter(img => (img.tags || []).includes(tag)));
};
```

У наведеній частині коду реалізовано клієнтську логіку пошуку та фільтрації зображень без повторного звернення до серверної частини. Пошук виконується у режимі реального часу за назвою та описом зображення, що забезпечує швидку реакцію інтерфейсу. Фільтрація за тегами дозволяє користувачу обмежити перелік відображуваних зображень до певної категорії. Усі операції виконуються над локальним станом, що позитивно впливає на продуктивність застосунку.

Наступним кроком перейдемо до реалізації візуального представлення окремого зображення.

```
const renderItem = ({ item }) => (
  <TouchableOpacity style={styles.card}
    onPress={() => navigation.navigate('Details', { imageId: item.id, title: item.title, description: item.description })}>
    <Image source={{ uri: item.storagePath }} style={styles.image} resizeMode="cover" />
    <View style={styles.cardContent}>
```

```
    <Text style={styles.name}>{item.title}</Text>
    {item.description && <Text style={styles.description} numberOfLines={2}>{item.description}</Text>}
   </View>
  </TouchableOpacity>
);
```

У наведеній частині коду реалізовано компонент renderItem, який відповідає за відображення одного зображення у вигляді інтерактивної картки. Картка містить прев'ю зображення, його назву та скорочений опис. Натискання на картку ініціює перехід до сторінки детального перегляду, при цьому необхідні параметри передаються через механізм навігації. Такий підхід забезпечує чітке розділення логіки між екранами та зручну навігацію в межах застосунку.

Наступним кроком перейдемо до формування загальної структури інтерфейсу головної сторінки.

У наведеній частині коду реалізовано умовне формування інтерфейсу головної сторінки залежно від стану застосунку та результатів асинхронних запитів до серверної частини. На етапі завантаження даних використовується індикатор ActivityIndicator, що сигналізує про виконання HTTP-запиту та блокує взаємодію з інтерфейсом до його завершення. Додатково здійснюється перевірка наявності авторизованого користувача шляхом аналізу значення userEmail, що дозволяє обмежити доступ до функціоналу у разі відсутності активної сесії. Після успішного завантаження даних формується основна структура екрана, яка включає поле пошуку для динамічної фільтрації зображень, кнопку переходу до екрана додавання нового зображення та компонент FlatList для відображення списку отриманих записів. Такий підхід забезпечує коректну обробку станів інтерфейсу, оптимізовану роботу зі списками даних та актуальне оновлення вмісту головної сторінки відповідно до дій користувача.

```
if (loading) return <ActivityIndicator size="large" color="#2e7d32" />;
return (
  <View style={pageStyles.container}>
    <Text style={pageStyles.title}>Зображення</Text>
    <TextInput style={pageStyles.searchInput} placeholder="Пошук зображень..."
```

```
    value={search} onChangeText={handleSearchChange} />
   <TouchableOpacity style={pageStyles.addButton} onPress={() => navigation.navigate('AddImage')}>
    <Text style={pageStyles.addButtonText}>Додати зображення</Text>
   </TouchableOpacity>
   <FlatList data={filteredImages} keyExtractor={(item) => item.id.toString()} renderItem={renderItem} />
  </View>
);
```

Після реалізації клієнтської логіки головної сторінки перейдемо до програмної реалізації серверної частини, яка забезпечує отримання та обробку даних про цифрові зображення. Серверні ендпоінти відповідають за взаємодію клієнтського застосунку з базою даних та реалізують механізми вибірки, фільтрації й логічного видалення зображень. Такий підхід дозволяє централізовано керувати даними та забезпечує коректну роботу клієнтського інтерфейсу. Першим кроком перейдемо до реалізації серверного ендпоінта, який відповідає за отримання списку зображень із бази даних. Даний маршрут використовується головною сторінкою застосунку для завантаження як власних зображень користувача, так і публічних зображень, доступних для перегляду. Вибір режиму роботи ендпоінта визначається параметрами HTTP-запиту.

```
app.get('/api/images', (req, res) => {
  const { userEmail, publicOnly } = req.query;
  let query = 'SELECT * FROM images WHERE is_deleted = 0';
  const params = [];
  if (publicOnly) query += ` AND visibility = 'public'`;
  else if (userEmail) { query += ' AND user_email = ?'; params.push(userEmail); }
  query += ' ORDER BY created_at DESC';
  db.all(query, params, (err, rows) => {
    if (err) return res.status(500).json({ error: 'Помилка сервера' });
    res.json(rows);
  });
});
```

У наведеному фрагменті коду реалізовано HTTP-ендпоінт типу GET /api/images, який забезпечує вибірку записів із таблиці images. Базовою умовою запиту є перевірка прапорця is_deleted, що дозволяє виключити з вибірки зображення, позначені як видалені. Такий підхід реалізує механізм логічного видалення та забезпечує збереження історичних даних у базі. Залежно від наявності параметрів userEmail або publicOnly, SQL-запит динамічно модифікується. У разі передавання параметра publicOnly вибірка обмежується

лише публічними зображеннями, що мають значення visibility = 'public'. Якщо ж передано userEmail, сервер повертає лише ті зображення, які належать конкретному користувачу. Параметризований запит із масивом params дозволяє запобігти SQL-ін’єкціям і підвищує рівень безпеки системи.

Результати вибірки впорядковуються за датою створення у зворотному порядку, що забезпечує відображення найновіших зображень першими на головній сторінці застосунку. Отримані дані повертаються клієнтській частині у форматі JSON та використовуються для формування динамічного списку зображень.

Наступним кроком перейдемо до реалізації серверного ендпоінта, який відповідає за видалення зображень. Видалення реалізовано у вигляді логічної операції, що дозволяє зберегти дані у базі та уникнути фізичного знищення записів. Такий підхід є доцільним для систем, що працюють з користувацьким контентом, і дозволяє забезпечити контроль доступу та можливість подальшого аналізу даних.

```
app.delete('/api/images/:id', (req, res) => {
  const { id } = req.params;
  const { userEmail } = req.query;
  db.run(
    'UPDATE images SET is_deleted = 1, deleted_at = CURRENT_TIMESTAMP WHERE id = ? AND user_email = ?',
    [id, userEmail],
    function (err) {
      if (err) return res.status(500).json({ error: 'Помилка сервера' });
      if (this.changes === 0) return res.status(403).json({ error: 'Видалення заборонено або зображення не знайдено'
});
      res.json({ success: true });
    }
  );
});
```

У наведеному фрагменті коду реалізовано HTTP-ендпоінт типу DELETE /api/images/:id, який виконує логічне видалення зображення за його ідентифікатором. Додатково перевіряється електронна пошта користувача, що ініціює видалення, що дозволяє гарантувати, що лише власник зображення має право на виконання цієї операції. Операція видалення виконується шляхом оновлення полів is_deleted та deleted_at, без фактичного видалення запису з

таблиці. Перевірка значення this.changes дозволяє визначити, чи було знайдено відповідний запис і чи мав користувач право на його видалення. У разі порушення умов сервер повертає відповідь з кодом 403, що відповідає стандартам REST API. Реалізація такого механізму забезпечує контроль доступу, цілісність даних та коректну взаємодію клієнтської і серверної частин застосунку.

Таким чином, у межах реалізації серверної частини головної сторінки створено набір ендпоінтів, що забезпечують отримання, фільтрацію та логічне видалення цифрових зображень. Серверна логіка тісно інтегрується з клієнтською частиною застосунку та забезпечує централізоване управління даними. Застосування параметризованих SQL-запитів, логічного видалення та перевірки прав доступу дозволяє підвищити безпеку, надійність і масштабованість розроблюваної системи управління цифровими зображеннями. Результат реалізації головної сторінки мобільного застосунку управління цифровими зображеннями наведено на рисунку 3.4.

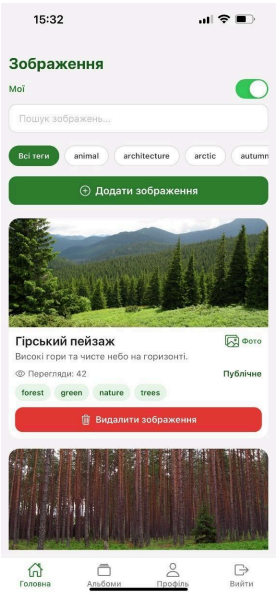

Рисунок 3.4. Інтерфейс головної сторінки мобільного застосунку управління цифровими зображеннями

Таким чином, у межах даного етапу реалізовано повноцінну головну сторінку застосунку, яка забезпечує перегляд, пошук, фільтрацію та видалення

цифрових зображень. Реалізований підхід поєднує ефективну клієнтську логіку з серверною обробкою даних.

3.3.4 Програмна реалізація логіки додавання нових зображень

Наступним кроком перейдемо до реалізації логіки додавання нових зображень у мобільному застосунку управління цифровими зображеннями. Основною метою цього етапу є забезпечення можливості користувачу вибрати файл із галереї, заповнити метадані зображення (назва, опис, теги), визначити режим доступу (публічне/приватне/за посиланням) та виконати завантаження на сервер із подальшим збереженням інформації у базі даних. Реалізація даного функціоналу включає клієнтську частину (інтерфейс і формування multipart/form-data запиту) та серверну частину (прийом файлу через multer, валідацію даних, запис у таблицю images і формування тегів у зв’язці tags + image_tags). Перейдемо до поетапної реалізації логіки даного функціоналу.

```
const AddImagePage = () => {
  const navigation = useNavigation();
  const [title, setTitle] = useState('');
  const [description, setDescription] = useState('');
  const [tagsText, setTagsText] = useState('');
  const [visibility, setVisibility] = useState('public');
  const [imageUri, setImageUri] = useState(null);
  const [submitting, setSubmitting] = useState(false);
};
```

У наведеній частині коду реалізовано ініціалізацію екрана додавання зображення у вигляді функціонального компонента AddImagePage. За допомогою хуків useState створюються змінні стану, які відповідають за введені користувачем метадані (title, description, tagsText), режим доступу (visibility), локальний шлях до вибраного зображення (imageUri) та прапорець submitting, що використовується для блокування повторних відправок під час виконання запиту. Додатково ініціалізується navigation, який застосовується для повернення на попередній екран після успішного створення нового запису.

Наступним кроком перейдемо до реалізації логіки вибору зображення з галереї мобільного пристрою.

```
const pickImage = async () => {
  const permission = await ImagePicker.requestMediaLibraryPermissionsAsync();
  if (!permission.granted) return Alert.alert('Помилка', 'Потрібен доступ до галереї');
  const result = await ImagePicker.launchImageLibraryAsync({
    mediaTypes: ImagePicker.MediaTypeOptions.Images,
    allowsEditing: true,
    quality: 0.8,
  });
  if (!result.canceled && result.assets?.length > 0) setImageUri(result.assets[0].uri);
};
```

У наведеній частині коду реалізовано процес вибору зображення на стороні клієнта за допомогою бібліотеки expo-image-picker. Спочатку виконується запит дозволу на доступ до медіатеки (requestMediaLibrary PermissionsAsync()), що є обов’язковим етапом для коректної роботи з файлами на мобільних ОС. У разі відсутності дозволу система припиняє виконання сценарію та виводить повідомлення про помилку. Далі викликається launchImageLibraryAsync(), який відкриває системну галерею, обмежує вибір лише зображеннями та дозволяє базове редагування (обрізання) перед завантаженням. Після успішного вибору зображення його локальний URI зберігається у стані imageUri, що надалі використовується під час формування multipart запиту на сервер.

Наступним кроком перейдемо до реалізації логіки відправлення даних на сервер та формування multipart/form-data запиту.

```
const handleCreate = async () => {
  if (submitting) return;
  if (!title.trim()) return Alert.alert('Помилка', 'Назва зображення є обов’язковою.');
  if (!imageUri) return Alert.alert('Помилка', 'Оберіть зображення.');
  try {
    setSubmitting(true);
    const userEmail = await AsyncStorage.getItem('userToken');
    if (!userEmail) return Alert.alert('Помилка', 'Користувач не авторизований');
    const formData = new FormData();
    ['userEmail', 'title', 'description', 'visibility', 'tags'].forEach(k => formData.append(k, eval(k)));
    formData.append('image', { uri: imageUri, name: 'upload.jpg', type: 'image/jpeg' });
    const res = await fetch(`${BASE_URL}/api/images/upload`, { method: 'POST', body: formData });
    // обробка відповіді...
  } finally { setSubmitting(false); }
};
```

У наведеній частині коду реалізовано основний клієнтський сценарій додавання зображення: валідацію форми, формування тіла запиту та обробку відповіді сервера. На початку виконується перевірка прапорця submitting, що запобігає повторним натисканням кнопки й паралельним запитам. Далі здійснюється базова перевірка обов’язкових полів: наявність назви (title) та вибраного файлу (imageUri). Після цього із AsyncStorage зчитується токен користувача (email), який використовується як ідентифікатор власника зображення на сервері. Для передачі файлу та додаткових полів формується об’єкт FormData, до якого додаються всі метадані та сам файл зображення як поле image. Запит надсилається на ендпоінт POST /api/images/upload у форматі multipart/form-data, що є стандартним підходом для завантаження файлів. У разі успіху система виводить повідомлення та повертається на попередній екран, а при помилці - відображає текст, який повернув сервер. Завершальним етапом у finally виконується скидання submitting, що відновлює можливість повторної взаємодії з формою.

Наступним кроком перейдемо до реалізації серверної частини прийому зображення та збереження файлу у файлову систему.

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => cb(null, uploadDir),
  filename: (req, file, cb) =>
    cb(null, `${Date.now()}-${Math.round(Math.random() * 1e9)}${path.extname(file.originalname)}`),
});
const upload = multer({
  storage,
  limits: { fileSize: 15 * 1024 * 1024 },
  fileFilter: (req, file, cb) => {
    const allowed = ['image/jpeg', 'image/png', 'image/webp'];
    allowed.includes(file.mimetype) ? cb(null, true) : cb(new Error('Дозволені формати: JPG, PNG, WEBP'));
  },
});
```

У наведеній частині коду реалізовано конфігурацію завантаження файлів на сервері за допомогою middleware multer. Використовується режим diskStorage, який зберігає файл у визначену директорію uploadDir. Для уникнення конфліктів імен формується унікальна назва файлу, що включає

часову мітку та випадковий суфікс із збереженням оригінального розширення. Додатково встановлено обмеження на розмір файлу (до 15 МБ) та реалізовано фільтрацію за MIME-типом, що дозволяє приймати лише коректні формати зображень. Такий підхід одночасно забезпечує безпечне завантаження та мінімізує ризик запису небажаних файлів.

Наступним кроком перейдемо до реалізації серверного ендпоінта, який приймає multipart-запит, виконує валідацію полів та створює новий запис у таблиці images.

```
app.post('/api/images/upload', upload.single('image'), async (req, res) => {
  const { userEmail, title, description, visibility, tags } = req.body;
  if (!userEmail) return res.status(400).json({ error: 'userEmail є обов’язковим' });
  if (!title) return res.status(400).json({ error: 'title є обов’язковим' });
  if (!req.file) return res.status(400).json({ error: 'Файл є обов’язковим' });
  const safeVis = ['public','private','link'].includes(visibility) ? visibility : 'public';
  const shareToken = safeVis === 'link' ? makeShareToken() : null;
  const storagePath = `${req.protocol}://${req.get('host')}/uploads/images/${req.file.filename}`;
  const insert = await runAsync(
    'INSERT INTO images
(user_email,title,description,original_filename,mime_type,size_bytes,storage_path,thumbnail_path,visibility,share_toke
n) VALUES (?,?,?,?,?,?,?,?,?,?)',
    [userEmail, title, description||null, req.file.originalname, req.file.mimetype, req.file.size, storagePath, null, safeVis,
shareToken]
  );
  // обробка тегів...
});
```

У наведеній частині коду реалізовано серверний маршрут POST /api/images/upload, який обробляє multipart/form-data запит, приймає файл та метадані зображення й виконує збереження у базі даних. Використання upload.single('image') забезпечує автоматичне витягування файлу з поля image та запис його у файлову систему відповідно до налаштувань multer. На початку маршруту проводиться валідація критичних даних: наявність userEmail, title та факту завантаження файлу. Далі нормалізується значення visibility, щоб уникнути некоректних режимів доступу, а при виборі режиму link автоматично генерується shareToken, який може бути використаний для доступу за посиланням. Після збереження файлу формується публічний шлях storagePath, який зберігається у таблиці images. Далі виконується SQL-вставка збережених параметрів, включно з технічними метаданими файлу (mime, size,

original_filename). Після створення запису додатково обробляються теги: рядок tags перетворюється у список значень, які зберігаються у таблиці tags та прив’язуються через таблицю зв’язку image_tags. Завершальним етапом сервер формує DTO нового зображення та повертає його клієнту разом із відсортованим списком тегів.

Наступним кроком перейдемо до реалізації серверної логіки обробки тегів та зв’язування їх із зображенням у моделі many-to-many.

```
const parseTags = (raw) => !raw ? [] :
  String(raw).split(',').map(t => t.trim()).filter(Boolean).slice(0, 30);

const setImageTags = async (imageId, tags) => {
  const clean = [...new Set(tags.map(t => t.toLowerCase()))].slice(0, 30);
  await runAsync('DELETE FROM image_tags WHERE image_id = ?', [imageId]);
  for (const name of clean) {
    await runAsync('INSERT OR IGNORE INTO tags (name) VALUES (?)', [name]);
    const tag = await getAsync('SELECT id FROM tags WHERE name = ?', [name]);
    if (tag) await runAsync('INSERT OR IGNORE INTO image_tags (image_id, tag_id) VALUES (?,?)', [imageId,
tag.id]);
  }
};
```

У наведеній частині коду реалізовано обробку тегів, яка забезпечує коректну підтримку зв’язку many-to-many між зображеннями та тегами. Функція parseTags() нормалізує вхідні дані, оскільки теги можуть надходити як рядок, розділений комами, або як масив значень. Додатково застосовується обмеження кількості тегів до 30, що запобігає зловживанням і перевантаженню системи. Функція setImageTags() виконує повну синхронізацію тегів для конкретного зображення: спочатку видаляються старі зв’язки з таблиці image_tags, після чого для кожного тегу виконується вставка у таблицю tags з використанням INSERT OR IGNORE, що дозволяє уникнути дублювання. Далі отримується id тегу й створюється запис у таблиці зв’язку. Таким чином, серверна частина забезпечує нормалізоване збереження тегів, повторне використання вже існуючих значень та коректне встановлення зв’язків між сутностями.

Результат реалізації інтерфейсу додавання нового зображення наведено на рисунку 3.5.

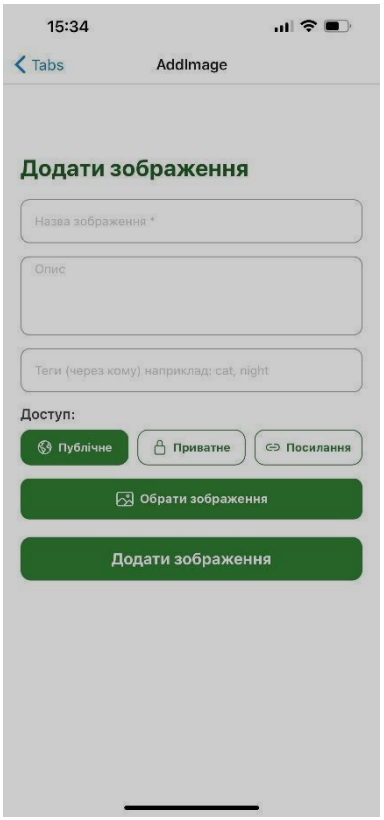

Рисунок 3.5. Інтерфейс додавання нового зображення у реалізованому мобільному застосунку

Таким чином, на даному етапі реалізовано повний цикл додавання зображення: від вибору файлу та введення метаданих у клієнтському інтерфейсі до збереження файлу на сервері, створення запису у таблиці images та формування системи тегів через зв’язок many-to-many. Реалізований механізм забезпечує валідацію даних, підтримку різних режимів доступу, а також підготовку структури для подальшого розширення функцій (наприклад, генерації thumbnail або додаткових метаданих про зображення).

3.3.5 Програмна реалізація логіки взаємодії з обраним зображенням

Наступним кроком перейдемо до реалізації логіки взаємодії користувача з обраним зображенням на сторінці детального перегляду. Даний екран є центральним елементом взаємодії із контентом, оскільки саме тут користувач отримує повну інформацію про зображення та виконує основні дії: перегляд метаданих (автор, дата додавання, кількість переглядів, теги), зміна приватності

(для власника), виставлення оцінки, додавання зображення до альбому, завантаження файлу на пристрій, а також перегляд і публікація коментарів. Реалізація включає формування запитів до серверної частини для отримання актуальних даних та виконання дій над ресурсом, збереження цих даних у стані компонента і відображення змін в інтерфейсі. Перейдемо до поетапної реалізації логіки даного функціоналу.

```
const DetailPage = ({ route }) => {
  const imageId = route?.params?.imageId;
  const [userEmail, setUserEmail] = useState('');
  const [loading, setLoading] = useState(true);
  const [image, setImage] = useState(null);
  const [comments, setComments] = useState([]);
  const [commentText, setCommentText] = useState('');
  const [myRating, setMyRating] = useState(0);
  const [albums, setAlbums] = useState([]);
  const isOwner = useMemo(() => image?.userEmail === userEmail, [image, userEmail]);
  const isPublic = useMemo(() => image?.visibility === 'public', [image]);
};
```

У наведеній частині коду реалізовано створення екрана DetailPage та ініціалізацію змінних стану, які зберігають дані для відображення і взаємодії. Значення imageId отримується з параметрів навігації та використовується як ключ для всіх API-запитів до сервера. Стан image містить повну модель зображення (шлях, опис, перегляди, теги, рейтинг тощо), а comments - масив коментарів, отриманих окремим запитом. Змінні myRating та savingRating відповідають за введення оцінки та блокування повторних натискань під час збереження рейтингу. Для операцій з альбомами передбачено окремі стани, які керують модальним вікном та процесом додавання зображення. Обчислювані властивості isOwner та isPublic реалізовано через useMemo, що дозволяє оптимізувати повторні рендери та відображати елементи керування (наприклад, перемикач приватності) лише коли це дозволено правами доступу.

Наступним кроком перейдемо до реалізації завантаження даних користувача та отримання інформації про зображення і коментарі з сервера.

```
const fetchImage = useCallback(async (viewerEmail) => {
  if (!imageId) return;
```

```
  const res = await fetch(`${BASE_URL}/api/images/${imageId}?viewerEmail=${encodeURIComponent(viewerEmail
|| '')}`);
  const data = await res.json();
  if (!res.ok) throw new Error(data?.error || 'Не вдалося отримати зображення');
  setImage(data);
  if (data.myRating != null) setMyRating(Number(data.myRating));
}, [imageId]);

const fetchComments = useCallback(async () => {
  if (!imageId) return;
  const data = await (await fetch(`${BASE_URL}/api/images/${imageId}/comments`)).json();
  setComments(Array.isArray(data) ? data : []);
}, [imageId]);
```

У наведеній частині коду реалізовано повний цикл завантаження даних для детальної сторінки: зчитування email користувача, отримання даних зображення та отримання коментарів. Функція loadUser() зчитує userToken з локального сховища AsyncStorage, зберігає його у стані userEmail і повертає як результат для подальшого використання. Функція fetchImage() формує запит до ендпоінта GET /api/images/:id, передаючи viewerEmail як параметр доступу - це дозволяє серверу повернути дані з урахуванням прав перегляду та, за необхідності, повернути оцінку поточного користувача (myRating). Отримані дані зберігаються в image, а поле myRating синхронізується зі станом для коректного відображення обраних зірок. Функція fetchComments() окремо завантажує список коментарів із маршруту GET /api/images/:id/comments і нормалізує відповідь до масиву.

Наступним кроком перейдемо до реалізації логіки зміни приватності зображення власником.

```
const toggleVisibility = async () => {
  if (!isOwner || !image) return;
  const nextVisibility = isPublic ? 'private' : 'public';
  try {
    const res = await fetch(`${BASE_URL}/api/images/${image.id}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ userEmail, visibility: nextVisibility }),
    });
    if (!res.ok) return Alert.alert('Помилка', (await res.json().catch(()=>({}))).error);
    await fetchImage(userEmail);
  } catch { Alert.alert('Помилка', 'Проблема з підключенням до сервера'); }
};
```

У наведеній частині коду реалізовано сценарій зміни видимості зображення, який доступний виключно власнику. Перед виконанням операції перевіряється умова isOwner, що унеможливлює зміну приватності іншими користувачами на рівні клієнта. Далі формується нове значення visibility шляхом перемикання між public і private. Для збереження змін виконується запит PUT /api/images/:id, у тілі якого передається userEmail як ідентифікатор власника та новий режим доступу. Після успішної відповіді викликається fetchImage(userEmail), щоб синхронізувати стан зображення з поточними даними з сервера і відобразити зміни в інтерфейсі без перезавантаження сторінки.

Наступним кроком перейдемо до реалізації логіки виставлення оцінки та оновлення середнього рейтингу.

```
const submitRating = async (value) => {
  if (!userEmail || !imageId) return Alert.alert('Помилка', 'Користувач не авторизований');
  setMyRating(value); setSavingRating(true);
  try {
    const res = await fetch(`${BASE_URL}/api/images/${imageId}/rating`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ userEmail, value }),
    });
    if (!res.ok) return Alert.alert('Помилка', (await res.json().catch(()=>({}))).error);
    await fetchImage(userEmail);
  } catch { Alert.alert('Помилка', 'Проблема з підключенням до сервера'); }
  finally { setSavingRating(false); }};
```

У наведеній частині коду реалізовано механізм встановлення або оновлення оцінки зображення. На початку перевіряється факт авторизації користувача (наявність userEmail), оскільки рейтинг прив’язується до конкретного облікового запису. Після цього значення оцінки одразу зберігається у стан myRating для миттєвого відображення в інтерфейсі, а прапорець savingRating вмикає режим блокування повторних дій. Запит надсилається на ендпоінт POST /api/images/:id/rating, який на сервері виконує upsert (оновлення або створення оцінки) для зв’язки «користувач-зображення». Після успішного збереження викликається fetchImage(userEmail), щоб оновити середній рейтинг (avgRating) і кількість оцінок (ratingsCount) відповідно до актуальних даних бази.

Наступним кроком перейдемо до реалізації додавання зображення до альбому через модальне вікно.

```
const openAlbumModal = async () => {
  if (!userEmail) return Alert.alert('Помилка', 'Користувач не авторизований');
  await fetch(`${BASE_URL}/api/albums?userEmail=${encodeURIComponent(userEmail)}`).then(r => r.json())
    .then(d => { setAlbums(Array.isArray(d) ? d : []); setSelectedAlbumId(null); setAlbumModalVisible(true); })
    .catch(() => Alert.alert('Помилка', 'Не вдалося завантажити альбоми'));
};
const addToAlbum = async () => {
  if (!selectedAlbumId) return Alert.alert('Помилка', 'Оберіть альбом');
  setAddingToAlbum(true);
  const res = await fetch(`${BASE_URL}/api/albums/${selectedAlbumId}/images`, { method: 'POST', headers: {
'Content-Type': 'application/json' }, body: JSON.stringify({ userEmail, imageId }) });
  if (!res.ok) Alert.alert('Помилка', (await res.json().catch(()=>({}))).error || 'Не вдалося додати до альбому');
};
```

У наведеній частині коду реалізовано механізм додавання поточного зображення до одного з альбомів користувача. Функція openAlbumModal() виконує запит GET /api/albums, передаючи userEmail як параметр вибірки - таким чином користувач бачить лише власні альбоми. Отримані дані записуються у стан albums, скидається попередній вибір selectedAlbumId, після чого відкривається модальне вікно. Функція addToAlbum() забезпечує валідацію вибору альбому та виконує запит POST /api/albums/:id/images, передаючи userEmail і imageId у тілі запиту. На сервері цей маршрут перевіряє права доступу, наявність альбому та зображення і створює зв’язок у таблиці album_images. Прапорець addingToAlbum блокує повторні дії, а після успішного додавання модальне вікно закривається та користувач отримує підтвердження.

Наступним кроком перейдемо до реалізації логіки додавання коментаря та оновлення списку коментарів.

```
const postComment = async () => {
  if (!userEmail) return Alert.alert('Помилка', 'Користувач не авторизований');
  const text = commentText.trim();
  if (!text) return;
  Keyboard.dismiss();
  try {
    const res = await fetch(`${BASE_URL}/api/images/${imageId}/comments`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ userEmail, content: text }),
    });
    if (!res.ok) return Alert.alert('Помилка', (await res.json().catch(()=>({}))).error);
```

```
    setCommentText(''); await fetchComments();
  } catch { Alert.alert('Помилка', 'Проблема з підключенням до сервера'); }
};
```

У наведеній частині коду реалізовано логіку створення нового коментаря до зображення. Перед відправкою виконується перевірка авторизації користувача та нормалізація введеного тексту методом trim(), що усуває випадкові пробіли та запобігає відправці порожніх повідомлень. Далі клавіатура примусово закривається для покращення UX. Коментар надсилається на сервер через POST /api/images/:id/comments, де дані прив’язуються до конкретного imageId і userEmail. Після успішної відповіді очищується поле введення commentText та повторно завантажується список коментарів через fetchComments(), що забезпечує відображення нового запису без перезавантаження сторінки.

Наступним кроком перейдемо до реалізації завантаження файлу зображення на пристрій користувача.

```
const handleDownloadImage = async () => {
  try {
    const { status } = await MediaLibrary.requestPermissionsAsync();
    if (status !== 'granted') return Alert.alert('Доступ заборонено', 'Потрібен доступ до галереї');
    const fileUri = FileSystem.documentDirectory + `image_${image.id}.jpg`;
    const result = await FileSystem.downloadAsync(image.storagePath, fileUri);
    await MediaLibrary.saveToLibraryAsync(result.uri);
    Alert.alert('Готово', 'Зображення збережено у галереї');
  } catch { Alert.alert('Помилка', 'Не вдалося завантажити зображення'); }
};
```

У наведеній частині коду реалізовано сценарій локального завантаження зображення на мобільний пристрій. Першим етапом виконується запит дозволу через MediaLibrary.requestPermissionsAsync(), оскільки збереження файлів у галерею потребує явного дозволу користувача. Далі формується шлях fileUri у внутрішньому каталозі застосунку (documentDirectory) із унікальною назвою, що прив’язана до image.id, що запобігає конфліктам при повторних завантаженнях. Метод FileSystem.downloadAsync() завантажує файл за URL image.storagePath та зберігає його локально, після чого MediaLibrary.saveToLibraryAsync() переносить файл у системну галерею,

роблячи його доступним у стандартних програмах перегляду. Такий підхід забезпечує коректну інтеграцію із файловою системою пристрою та дозволяє користувачу швидко зберегти потрібне зображення офлайн.

```
if (loading) {
   return <ActivityIndicator style={{ marginTop: 100 }} size="large" color="#2e7d32" />;
}

if (!image) {
   return (
      <View style={[styles.container, { justifyContent: 'center' }]}>
         <Text style={{ textAlign: 'center', fontSize: 16 }}>Зображення не знайдено</Text>
      </View>
   );
}
```

У наведеній частині коду реалізовано базову перевірку станів завантаження та доступності даних. При виконанні асинхронних операцій (отримання зображення та коментарів) відображається індикатор ActivityIndicator, що сигналізує користувачу про процес завантаження. Після завершення запитів виконується перевірка image: якщо сервер повернув помилку або ресурс не існує, відображається альтернативний екран з повідомленням. Така структура є необхідною для уникнення помилок рендерингу (коли інтерфейс намагається звернутися до полів image, які ще не визначені) і забезпечує стабільну роботу сторінки за різних сценаріїв доступу та мережевих збоїв.

Результат реалізації інтерфейсу взаємодії з обраним зображенням наведено на рисунках 3.6 та 3.7.

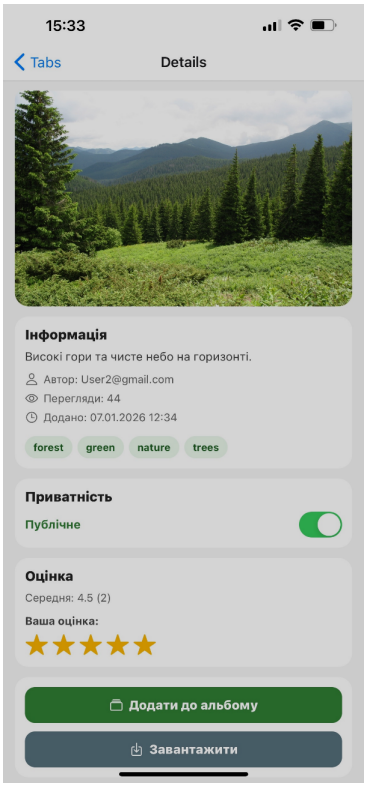

Рисунок 3.6. Інтерфейс детальної сторінки зображення у мобільному застосунку

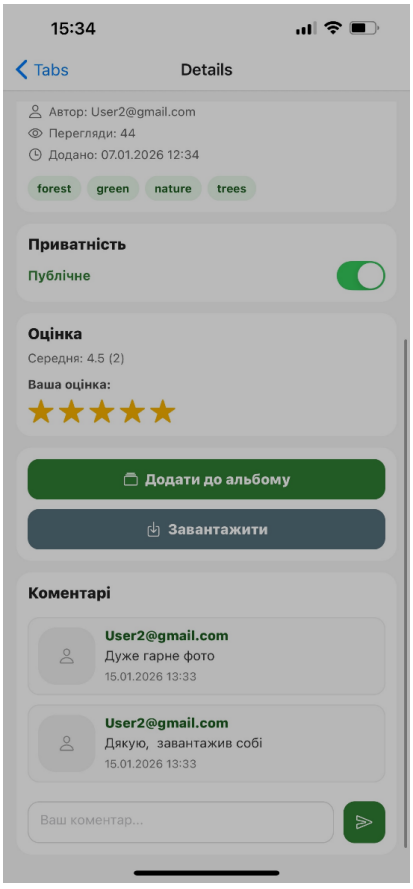

Рисунок 3.7. Інтерфейс детальної сторінки зображення у мобільному застосунку

Таким чином, на даному етапі реалізовано комплексну сторінку взаємодії з обраним зображенням, яка об’єднує отримання детальних даних із сервера та виконання прикладних дій над ресурсом: зміна приватності, оцінювання, коментування, додавання до альбомів і локальне завантаження. Реалізований підхід забезпечує синхронізацію клієнтського стану з серверними даними після кожної операції та підтримує коректну роботу механізмів прав доступу, що є ключовим для системи управління цифровими зображеннями.

3.3.6 Програмна реалізація логіки взаємодії з альбомами користувача

Наступним кроком перейдемо до реалізації логіки взаємодії користувача з альбомами, які застосовуються для структурування цифрових зображень у тематичні колекції. На даній сторінці користувач повинен мати можливість переглядати список власних альбомів із кількістю зображень та обкладинкою, відкривати вибраний альбом для детального перегляду його вмісту, створювати нові альбоми через форму у модальному вікні, а також видаляти непотрібні альбоми. Реалізація даного функціоналу передбачає завантаження даних з серверної частини за email авторизованого користувача, підтримку повторного оновлення списку при поверненні на екран, а також виконання POST/DELETE-запитів для створення та видалення альбомів. Перейдемо до поетапної реалізації логіки даного функціоналу.

```
const AlbumsPage = ({ navigation }) => {
  const [albums, setAlbums] = useState([]);
  const [modalVisible, setModalVisible] = useState(false);
  const [newName, setNewName] = useState('');
  const [newDescription, setNewDescription] = useState('');
  const [userEmail, setUserEmail] = useState('');
  useEffect(() => {
    AsyncStorage.getItem('userToken').then(email => {
      setUserEmail(email || '');
      if (email) fetchAlbums(email); else setAlbums([]);
    });
  }, []);
};
```

У наведеній частині коду реалізовано ініціалізацію ключових змінних стану, необхідних для роботи екрана альбомів. Масив albums використовується

як основне джерело даних для формування списку, а modalVisible керує відображенням модального вікна створення нового альбому. Поля newName та newDescription зберігають введені користувачем значення форми, що надалі передаються на сервер під час створення альбому. Додатково ініціалізовано стан userEmail, який визначає, які саме альбоми потрібно завантажувати з бази даних. У блоці useEffect реалізовано асинхронне зчитування userToken з AsyncStorage, що забезпечує прив'язку даних до поточного користувача. Якщо email відсутній, список альбомів очищується, щоб уникнути відображення сторонніх або застарілих даних. Хук useFocusEffect застосовано для повторного завантаження альбомів при кожному поверненні на екран, що особливо важливо після додавання зображень до альбомів або створення нового альбому на інших екранах.

Наступним кроком перейдемо до реалізації отримання списку альбомів з серверної частини та збереження результату в стані компонента.

```
const fetchAlbums = async (email) => {
  try {
    const res = await fetch(`${BASE_URL}/api/albums?userEmail=${encodeURIComponent(email)}`);
    const data = await res.json();
    setAlbums(Array.isArray(data) ? data : []);
  } catch { setAlbums([]); }
};

const openModal = () => setModalVisible(true);
const closeModal = () => { Keyboard.dismiss(); setModalVisible(false); };
```

У наведеній частині коду реалізовано отримання списку альбомів користувача через ендпоінт GET /api/albums. Email передається як query-параметр userEmail, що дозволяє серверу виконати вибірку лише тих альбомів, які належать поточному користувачу. Для коректного формування URL використано encodeURIComponent, що запобігає помилкам при наявності спеціальних символів у пошті. Отримана відповідь перетворюється в JSON і нормалізується перевіркою Array.isArray(data), щоб у стан albums завжди потрапляв масив (навіть якщо сервер поверне неочікуваний формат). У разі

мережевої помилки або недоступності сервера список очищується, а помилка фіксується у консолі, що спрощує діагностику під час тестування.

Наступним кроком перейдемо до реалізації логіки керування модальним вікном створення альбому.

```
const closeModal = () => {
   Keyboard.dismiss();
   setModalVisible(false);
};

const openModal = () => {
   setModalVisible(true);
};
```

У наведеній частині коду реалізовано допоміжні функції керування модальним вікном. Функція openModal() активує відображення форми створення альбому, змінюючи стан modalVisible. Функція closeModal() додатково виконує Keyboard.dismiss(), що примусово закриває клавіатуру перед закриттям модального вікна. Такий підхід усуває типову проблему мобільних інтерфейсів, коли клавіатура може залишатися відкритою або спричиняти некоректні відступи після закриття модального шару.

Наступним кроком перейдемо до реалізації створення нового альбому з відправленням даних на сервер.

```
const createAlbum = async () => {
  Keyboard.dismiss();
  if (!newName.trim()) return Alert.alert('Помилка', 'Введіть назву альбому');
  try {
    const res = await fetch(`${BASE_URL}/api/albums`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ userEmail, name: newName.trim(), description: newDescription.trim() || null, visibility:
'public' }),
    });
    if (!res.ok) return Alert.alert('Помилка', (await res.json().catch(()=>({}))).error);
    setNewName(''); setNewDescription(''); closeModal(); fetchAlbums(userEmail);
  } catch { Alert.alert('Помилка', 'Проблема з підключенням до сервера'); }
};
```

У наведеній частині коду реалізовано сценарій створення нового альбому. На початковому етапі виконується нормалізація введення: назва перевіряється

через trim(), що не дозволяє створити альбом із порожнім або “пробільним” іменем. Далі формується запит POST /api/albums, у якому передається userEmail для встановлення власника альбому, name як обов’язкове поле, description як необов’язкове поле (за відсутності передається null), а також параметр visibility. Після отримання відповіді виконується перевірка res.ok: у випадку помилки користувач отримує повідомлення з текстом data.error, який повертається сервером. Якщо альбом створено успішно, поля форми очищуються, модальне вікно закривається, а список альбомів повторно завантажується через fetchAlbums(userEmail), що гарантує синхронізацію інтерфейсу з актуальним станом бази даних.

Наступним кроком перейдемо до реалізації логіки видалення альбому з підтвердженням дії.

```
const deleteAlbum = (albumId) =>
  Alert.alert('Видалити альбом', 'Ви впевнені?', [
    { text: 'Скасувати', style: 'cancel' },
    { text: 'Видалити', style: 'destructive', onPress: async () => {
      try {
        const res = await
fetch(`${BASE_URL}/api/albums/${albumId}?userEmail=${encodeURIComponent(userEmail)}`, { method:
'DELETE' });
        if (!res.ok) return Alert.alert('Помилка', (await res.json().catch(()=>({}))).error);
        fetchAlbums(userEmail);
      } catch { Alert.alert('Помилка', 'Не вдалося видалити альбом'); }
    }},
  ]);
```

У наведеній частині коду реалізовано безпечне видалення альбому з обов’язковим підтвердженням користувача. Для цього використано системний діалог Alert.alert, який пропонує два варіанти: скасувати дію або підтвердити видалення. Реальне видалення відбувається лише після натискання кнопки “Видалити”, що знижує ризик випадкової втрати даних. Після підтвердження виконується запит DELETE /api/albums/:id, причому userEmail передається як query-параметр для перевірки прав доступу на сервері (сервер повинен переконатися, що альбом видаляє саме власник). У разі успішної відповіді список альбомів оновлюється повторним викликом fetchAlbums, що забезпечує

моментальне видалення елемента зі списку без ручного перезавантаження екрана.

Наступним кроком перейдемо до реалізації основної структури інтерфейсу сторінки - заголовка, списку альбомів, кнопки створення та модального вікна.

```
return ( <View style={styles.container}><Text style={styles.title}>Мої альбоми</Text>
  {albums.length === 0 ? <View style={styles.centeredTextContainer}><Text style={styles.emptyText}>У вас ще немає альбомів</Text></View> :
    <FlatList data={albums} keyExtractor={item => item.id.toString()} renderItem={renderItem} contentContainerStyle={{ paddingBottom: 10 }} keyboardShouldPersistTaps="handled" />}
  <TouchableOpacity style={styles.addButton} onPress={openModal}><Ionicons name="add-circle-outline" size={20} color="#fff" /><Text style={styles.addButtonText}>Додати альбом</Text></TouchableOpacity>
  <Modal visible={modalVisible} transparent animationType="slide" onRequestClose={closeModal}><TouchableWithoutFeedback onPress={Keyboard.dismiss}><View style={styles.modalOverlay}>
    <KeyboardAvoidingView behavior={Platform.OS === 'ios' ? 'padding' : undefined} keyboardVerticalOffset={Platform.OS === 'ios' ? 20 : 0}><TouchableWithoutFeedback><View style={styles.modalView}>
      <Text style={styles.label}>Створення альбому</Text><Text style={styles.fieldLabel}>Назва альбому</Text>
      <TextInput style={styles.input} placeholder="Напр.: Природа" value={newName} onChangeText={setNewName} />
      <Text style={styles.fieldLabel}>Опис</Text><TextInput style={[styles.input, styles.textArea]} placeholder="Коротко опишіть, що зберігається в альбомі (необов’язково)" value={newDescription} onChangeText={setNewDescription} multiline />
      <TouchableOpacity style={styles.saveButton} onPress={createAlbum}><Text style={styles.saveButtonText}>Зберегти</Text></TouchableOpacity>
      <TouchableOpacity style={styles.cancelButton} onPress={closeModal}><Text style={styles.cancelButtonText}>Скасувати</Text></TouchableOpacity>

</View></TouchableWithoutFeedback></KeyboardAvoidingView></View></TouchableWithoutFeedback></Modal></View> );
```

У наведеній частині коду реалізовано повну структуру екрана керування альбомами. Заголовок “Мої альбоми” задає контекст сторінки, після чого виконується умовне відображення: якщо масив albums порожній - показується інформаційне повідомлення, якщо дані присутні - використовується FlatList, що забезпечує ефективний рендеринг списку при великій кількості елементів. Кнопка “Додати альбом” відкриває модальне вікно створення, де користувач вводить назву та опис майбутньої колекції. Модальне вікно реалізовано через Modal із напівпрозорим overlay та KeyboardAvoidingView, що дозволяє коректно зміщувати форму при появі клавіатури (особливо на iOS) і не перекривати поля

введення. Натискання “Зберегти” запускає createAlbum(), а “Скасувати” закриває модальне вікно та очищає фокус із клавіатури.

Результат реалізації інтерфейсу взаємодії користувача з альбомами наведено на рисунку 3.8.

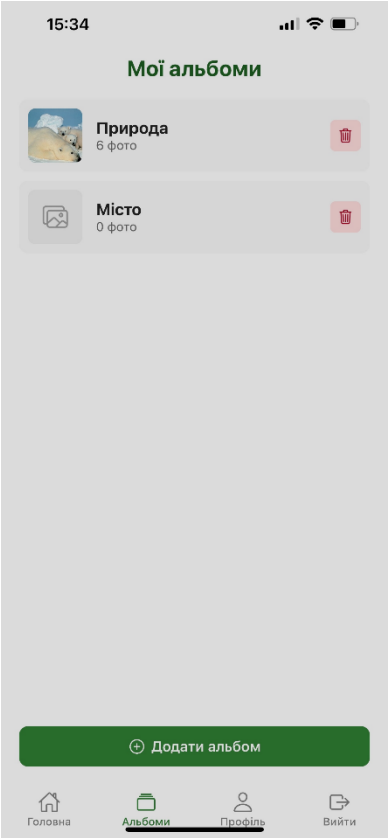

Рисунок 3.8. Інтерфейс сторінки керування альбомами у реалізованому мобільному застосунку

Таким чином, на даному етапі реалізовано повний цикл керування альбомами користувача: завантаження списку з сервера, відображення обкладинки та кількості зображень, створення альбомів через модальну форму, видалення з підтвердженням і перехід до сторінки детального перегляду альбому. Реалізована логіка забезпечує синхронізацію клієнтського інтерфейсу з серверною базою даних після кожної операції та створює основу для подальшого опрацювання функціоналу перегляду вмісту альбому на окремому екрані.

3.3.7 Програмна реалізація логіки взаємодії з обраним альбомом

Наступним кроком перейдемо до реалізації логіки взаємодії з обраним альбомом користувача. Основним призначенням даного екрана є відображення вмісту конкретного альбому у вигляді сітки зображень (по два елементи в рядку), забезпечення швидкого переходу до сторінки детального перегляду вибраного зображення, а також інформування користувача про порожній стан альбому. Реалізація передбачає отримання ідентифікатора альбому з параметрів навігації, виконання запиту до серверного API для завантаження списку зображень, збереження результату у стані компонента та повторне оновлення даних під час повернення на екран. Перейдемо до поетапної реалізації логіки даного функціоналу.

```
const DetailAlbumPage = () => {
   const route = useRoute();
   const navigation = useNavigation();
   const { albumId, name } = route.params;

   const [loading, setLoading] = useState(true);
   const [images, setImages] = useState([]);
};
```

У наведеній частині коду реалізовано підготовку екрана детального перегляду альбому. За допомогою useRoute() отримуються параметри навігації, з яких витягуються albumId (ключовий ідентифікатор для виконання серверних запитів) та name (назва альбому для відображення в заголовку сторінки). Також ініціалізовано стани loading та images: перший використовується для контролю відображення індикатора завантаження під час мережевих операцій, а другий - як контейнер для списку зображень, отриманих із серверної бази даних.

Наступним кроком перейдемо до реалізації завантаження зображень обраного альбому з серверної частини.

```
const fetchAlbumImages = useCallback(async () => {
   try {
     setLoading(true);
     const data = await (await fetch(`${BASE_URL}/api/albums/${albumId}/images`)).json().catch(() => []);
     setImages(Array.isArray(data) ? data : []);
```

```
    } catch { setImages([]); }
    finally { setLoading(false); }
  }, [albumId]);
};
```

У наведеній частині коду реалізовано асинхронну функцію fetchAlbumImages(), яка виконує запит до ендпоінта GET /api/albums/:id/images для отримання списку зображень, прив’язаних до конкретного альбому. Перед початком запиту стан loading переводиться у значення true, що дозволяє інтерфейсу коректно показувати користувачу процес завантаження. Після отримання відповіді застосовується безпечне перетворення у JSON з fallback-варіантом [], щоб уникнути аварійного завершення у випадку некоректної відповіді сервера. Далі дані нормалізуються перевіркою Array.isArray, що гарантує збереження у стані саме масиву, і відповідно коректну роботу FlatList. У блоці catch передбачено очищення images при помилці, щоб не залишати на екрані старі дані, а у finally - обов’язкове вимикання індикатора завантаження.

Наступним кроком перейдемо до реалізації візуального представлення одного елемента сітки зображень та переходу на сторінку детального перегляду.

```
const renderItem = ({ item }) => {
  const imgUri = item.thumbnailPath || item.storagePath;
  return (
    <TouchableOpacity activeOpacity={0.9} style={styles.card} onPress={() => navigation.navigate('Details', {
imageId: item.id, title: item.title, description: item.description })}>
      {imgUri ? <Image source={{ uri: imgUri }} style={styles.image} resizeMode="cover" /> :
        <View style={[styles.image, styles.placeholder]}><Ionicons name="image-outline" size={28} color="#9e9e9e"
/></View>}
      <View style={styles.cardFooter}>
        <Text style={styles.cardTitle} numberOfLines={1}>{item.title || 'Без назви'}</Text>
      </View>
    </TouchableOpacity>
  );
};
```

У наведеній частині коду реалізовано логіку відображення одного зображення в межах альбому. Спочатку визначається джерело для прев’ю: використовується thumbnailPath, якщо він доступний (що зменшує споживання трафіку та прискорює рендер), або storagePath, якщо мініатюра ще не

сформована. Далі формується клікабельна картка TouchableOpacity, натискання на яку виконує перехід на екран Details із передачею ключового параметра imageId. Передавання також title та description є допоміжним, однак базовим ідентифікатором для завантаження повних даних залишається саме imageId. Візуальна частина картки адаптована до ситуації, коли посилання на зображення відсутнє: у такому випадку замість фото відображається placeholder із піктограмою, що зберігає цілісність UI навіть при помилкових даних.

Наступним кроком перейдемо до реалізації загальної структури сторінки - заголовка, відображення індикатора завантаження, сітки зображень та порожнього стану.

```
return (
  <View style={styles.container}>
    <View style={styles.header}>
      <TouchableOpacity style={styles.backBtn} onPress={() => navigation.goBack()} activeOpacity={0.8}><Ionicons
name="chevron-back" size={22} color="#1b5e20" /></TouchableOpacity>
      <Text style={styles.headerTitle} numberOfLines={1}>{name}</Text><View style={styles.backBtn} />
    </View>
    {loading ? <ActivityIndicator style={{ marginTop: 40 }} size="large" color="#2e7d32" /> :
      <FlatList data={images} keyExtractor={item => item.id.toString()} renderItem={renderItem} numColumns={2}
columnWrapperStyle={styles.row} contentContainerStyle={styles.list} showsVerticalScrollIndicator={false}
        ListEmptyComponent={
          <View style={styles.emptyWrap}><Ionicons name="albums-outline" size={44} color="#9e9e9e" />
            <Text style={styles.emptyTitle}>В альбомі поки немає фото</Text><Text style={styles.emptySub}>Додайте
фото через DetailPage → “Додати до альбому”</Text></View>
        } />}</View>
);
```

У наведеній частині коду реалізовано побудову повного інтерфейсу екрана обраного альбому. У верхній частині формується кастомний заголовок із кнопкою повернення navigation.goBack() та назвою альбому, яка передається з попереднього екрана через параметри навігації. Додатковий порожній блок справа повторює розмір кнопки “назад”, що забезпечує симетрію та центрування заголовка незалежно від довжини назви. Далі реалізовано умовний рендеринг: якщо loading=true, показується ActivityIndicator, а після завершення запиту - FlatList, налаштований у режимі двох колонок (numColumns={2}) з використанням columnWrapperStyle для рівномірного розподілу карток. Окремо визначено ListEmptyComponent, який відображається у випадку порожнього

альбому та виконує роль підказки користувачу щодо механізму додавання зображень до альбому через сторінку DetailPage.

Після реалізації клієнтської частини даного функціоналу наступним кроком реалізуємо серверний ендпоінт отримання списку зображень у межах конкретного альбому, який використовується екраном DetailAlbumPage.

```
app.get('/api/albums/:id/images', async (req, res) => {
  try {
    const rows = await allAsync(
      `SELECT i.id, i.user_email AS userEmail, i.title, i.description,
            i.storage_path AS storagePath, i.thumbnail_path AS thumbnailPath,
            i.visibility, ai.position, ai.added_at AS addedAt
       FROM album_images ai JOIN images i ON i.id = ai.image_id
       WHERE ai.album_id = ? AND i.is_deleted = 0
       ORDER BY ai.position ASC, ai.added_at ASC`,
      [req.params.id]
    );
    const result = await Promise.all(rows.map(async r => ({ ...r, tags: await getImageTags(r.id) })));
    res.json(result);
  } catch (e) { res.status(500).json({ error: 'Помилка сервера' }); }
});
```

У наведеній частині коду реалізовано серверний маршрут GET /api/albums/:id/images, який повертає перелік зображень, пов’язаних із заданим альбомом через таблицю зв’язків album_images. Параметр id зчитується з req.params, після чого виконується SQL-запит з JOIN до таблиці images. Умова i.is_deleted = 0 гарантує, що soft-deleted зображення не потрапляють у результати, навіть якщо зв’язок з альбомом існує. Сортування реалізовано через ORDER BY ai.position ASC, ai.added_at ASC, що забезпечує стабільний порядок відображення відповідно до позиції зображення в альбомі та часу додавання. Після вибірки виконується додатковий крок - отримання тегів для кожного зображення через getImageTags(row.id), щоб клієнтська частина могла відображати або використовувати теги без додаткових запитів. У випадку помилки сервер повертає код 500 та повідомлення про помилку, що дозволяє клієнту коректно обробити збій.

Результат реалізації інтерфейсу взаємодії користувача з обраним альбомом наведено на рисунку 3.9.

Рисунок 3.9. Інтерфейс взаємодії користувача з обраним альбомом у реалізованому мобільному застосунку

Таким чином, на даному етапі реалізовано механізм перегляду вмісту обраного альбому: завантаження зображень із серверної бази даних, відображення їх у вигляді адаптивної сітки по два елементи в рядку, підтримку порожнього стану та навігацію до сторінки детального перегляду конкретного зображення. Використання useFocusEffect забезпечує актуальність даних при повторних переходах на екран, а серверний ендпоінт гарантує коректну вибірку лише активних (не видалених) зображень, пов’язаних із вибраним альбомом

3.3.8 Програмна реалізація логіки взаємодії з особистими даними користувача

Останнім кроком перейдемо до реалізації логіки взаємодії з особистими даними користувача. Метою даного етапу є забезпечення перегляду профілю авторизованого користувача, відображення базової інформації (ім’я, телефон,

вік, біографія), зміни аватара та редагування персональних даних із подальшим збереженням на сервері. Додатково екран профілю формує коротку статистику активності користувача (кількість зображень, альбомів та коментарів), що потребує виконання кількох паралельних запитів до API та об’єднання результатів на клієнті. Перейдемо до поетапної реалізації логіки даного функціоналу.

```
const ProfilePage = () => {
  const [loading, setLoading] = useState(true);
  const [userEmail, setUserEmail] = useState(null);
  const [profile, setProfile] = useState(null);
  const [stats, setStats] = useState({ imagesCount: 0, albumsCount: 0, commentsCount: 0 });
  const [isEditing, setIsEditing] = useState(false);
  const [editName, setEditName] = useState('');
  const [editPhone, setEditPhone] = useState('');
  const [editAge, setEditAge] = useState('');
  const [editBio, setEditBio] = useState('');
};
```

У наведеній частині коду реалізовано підготовку змінних стану для коректної роботи екрана профілю. Стан loading використовується для відображення індикатора під час мережевих запитів, а userEmail - як ідентифікатор поточного користувача, що зчитується з AsyncStorage. Об’єкт profile містить дані, отримані з серверного ендпоінта профілю, а структура stats агрегує статистичні показники, що завантажуються через окремі запити. Стан isEditing керує режимами “перегляд / редагування”, а поля editName, editPhone, editAge, editBio застосовуються як контрольовані значення форми редагування - це дозволяє відокремити введення користувача від збережених серверних даних і виконувати валідацію перед відправленням.

Наступним кроком перейдемо до реалізації завантаження профілю та статистики користувача з серверної частини.

```
const loadAll = useCallback(async () => {
  try { setLoading(true);
    const token = await AsyncStorage.getItem('userToken');
    if (!token) { setUserEmail(null); setProfile(null); return; }
    setUserEmail(token);
    const profileData = await fetch(`${BASE_URL}/api/profile?email=${encodeURIComponent(token)}`).then(r =>
r.json());
```

```
    setProfile(profileData);
    ['Name','Phone','Age','Bio'].forEach(f => eval(`setEdit${f}(profileData.${f.toLowerCase()} ?? '')`));
    const [img, alb, com] = await Promise.all([
      fetch(`${BASE_URL}/api/images?userEmail=${encodeURIComponent(token)}`).then(r => r.json()).catch(() =>
[]), fetch(`${BASE_URL}/api/albums?userEmail=${encodeURIComponent(token)}`).then(r => r.json()).catch(() =>
[]), fetch(`${BASE_URL}/api/comments/count?userEmail=${encodeURIComponent(token)}`).then(r =>
r.json()).catch(() => ({}))
    ]);
    setStats({ imagesCount: img.length, albumsCount: alb.length, commentsCount: com.count ?? 0 }); } finally {
setLoading(false); }
}, []);
```

У наведеній частині коду реалізовано централізовану функцію loadAll(), яка виконує повне завантаження даних для екрана профілю. Спочатку з локального сховища AsyncStorage зчитується userToken, який у межах застосунку виступає електронною поштою користувача. У разі відсутності токена система переводить екран у стан “не авторизований” та припиняє подальші запити. Далі виконується запит GET /api/profile для отримання персональних даних, а отриманий результат зберігається у profile. Паралельно відбувається ініціалізація контрольованих полів редагування (editName, editPhone, editAge, editBio) на основі серверних значень, щоб при переході у режим редагування користувач бачив актуальні дані. Окремим кроком виконується Promise.all, який дозволяє одночасно отримати списки зображень та альбомів, а також кількість коментарів - таким чином зменшується загальний час очікування, а статистика формується на клієнті через довжину масивів і числове поле count.

Наступним кроком перейдемо до реалізації функціоналу зміни аватара користувача.

```
const handleChangeAvatar = async () => {
  if (!userEmail) return;
  const { status } = await ImagePicker.requestMediaLibraryPermissionsAsync();
  if (status !== 'granted') return Alert.alert('Помилка', 'Потрібен доступ до галереї');
  const result = await ImagePicker.launchImageLibraryAsync({ mediaTypes: ImagePicker.MediaTypeOptions.Images,
quality: 0.7 });
  if (result.canceled || !result.assets?.length) return;
  const res = await fetch(`${BASE_URL}/api/profile`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ email: userEmail, avatarUrl: result.assets[0].uri }),
  });
  if (!res.ok) return Alert.alert('Помилка', (await res.json().catch(()=>({}))).error);
  await loadAll();};
```

У наведеній частині коду реалізовано зміну аватара користувача через інтеграцію з системною галереєю. Спочатку запитується дозвіл на доступ до медіатеки (requestMediaLibraryPermissionsAsync), і в разі відмови користувачу повертається повідомлення про неможливість продовження. Далі запускається вибір зображення launchImageLibraryAsync, після чого з результату береться uri вибраного файлу. Для збереження змін виконується запит PUT /api/profile, де передаються email користувача та новий avatarUrl. Після успішної відповіді викликається loadAll(), щоб синхронізувати локальний стан з оновленими серверними даними та одразу відобразити новий аватар на екрані.

Наступним кроком перейдемо до реалізації збереження змінених персональних даних користувача.

```
const saveProfile = async () => {
  if (!userEmail) return;
  Keyboard.dismiss();
  const ageNum = editAge.trim() ? Number(editAge.trim()) : null;
  if (ageNum && (!Number.isInteger(ageNum) || ageNum < 1 || ageNum > 120))
    return Alert.alert('Помилка', 'Вік має бути цілим числом 1..120');
  const res = await fetch(`${BASE_URL}/api/profile`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ email: userEmail, name: editName.trim()||null, phone: editPhone.trim()||null, age: ageNum,
bio: editBio.trim()||null }),
  });
  if (!res.ok) return Alert.alert('Помилка', (await res.json().catch(()=>({}))).error);
  setIsEditing(false); await loadAll();
};
```

У наведеній частині коду реалізовано збереження змін профілю з базовою валідацією введених даних. Перед відправленням запиту приховується клавіатура (Keyboard.dismiss()), після чого відбувається перевірка поля віку: значення перетворюється у число та контролюється, щоб воно було цілим у межах 1..120. Далі формується запит PUT /api/profile, у якому передаються лише актуальні значення полів: рядкові параметри нормалізуються через trim() та передаються як null, якщо користувач залишив їх порожніми. Після успішного збереження вимикається режим редагування (setIsEditing(false)) і виконується

повторне завантаження даних через loadAll() для синхронізації стану та оновлення відображення.

Після реалізації клієнтської частини наступним кроком реалізуємо серверні ендпоінти, які забезпечують отримання та оновлення персональних даних користувача.

```
app.get('/api/profile', async (req, res) => {
  const { email } = req.query;
  if (!email) return res.status(400).json({ error: 'email є обов’язковим' });
  const user = await getAsync(
    'SELECT id, name, phone, age, email, avatar_url AS avatarUrl, bio, created_at AS createdAt FROM users WHERE
email = ?', [email]
  );
  if (!user) return res.status(404).json({ error: 'Користувача не знайдено' });
  res.json(user);
});
```

У наведеній частині коду реалізовано маршрут GET /api/profile, який повертає профіль користувача за параметром email. На сервері виконується перевірка наявності обов’язкового параметра, після чого здійснюється вибірка з таблиці users. Для зручності клієнта застосовано перейменування полів (наприклад, avatar_url → avatarUrl), що дозволяє використовувати дані без додаткової трансформації на фронтенді. Якщо користувача не знайдено, повертається код 404, а у разі внутрішньої помилки - 500 з повідомленням про збій.

```
app.put('/api/profile', async (req, res) => {
  const { email, name, phone, age, avatarUrl, bio } = req.body;
  if (!email) return res.status(400).json({ error: 'email є обов’язковим' });
  const r = await runAsync(
    `UPDATE users SET name=COALESCE(?,name), phone=?, age=?, avatar_url=?, bio=?, updated_at=datetime('now')
WHERE email=?`,
    [name||null, phone||null, age||null, avatarUrl||null, bio||null, email]
  );
  if (r.changes === 0) return res.status(404).json({ error: 'Користувача не знайдено' });
  res.json({ success: true });
});
```

У наведеній частині коду реалізовано маршрут PUT /api/profile, який відповідає за оновлення персональних даних користувача у таблиці users. Сервер отримує дані з тіла запиту та виконує перевірку наявності email, який

визначає запис для оновлення. Оновлення реалізовано через SQL-вираз UPDATE, де updated_at встановлюється автоматично на поточний час, а значення полів передаються з нормалізацією null. Для поля name використано COALESCE, що дозволяє не перезаписувати ім’я, якщо воно не було передане клієнтом. Після виконання запиту перевіряється кількість змінених рядків (changes): якщо змін немає, це означає, що користувача не знайдено.

Результат реалізації інтерфейсу взаємодії з особистими даними користувача наведено на рисунку 3.9.

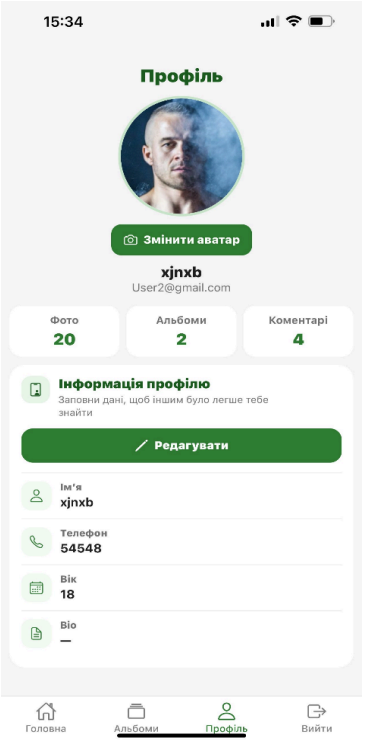

Рисунок 3.9. Інтерфейс взаємодії з особистими даними користувача у реалізованому мобільному застосунку

Таким чином, на даному етапі реалізовано повноцінну логіку роботи з персональними даними користувача: завантаження профілю з сервера, обчислення статистики активності на основі паралельних запитів, перемикання режимів перегляду й редагування, валідацію введених даних та збереження змін через серверний API. Це забезпечує персоналізацію застосунку та підтримує коректну синхронізацію даних між клієнтською і серверною частинами системи.

Висновки до розділу 3

У третьому розділі було виконано програмну реалізацію мобільного застосунку для управління цифровими зображеннями відповідно до спроєктованої архітектури та визначених функціональних вимог. Послідовно реалізовано клієнтську та серверну частини системи, що забезпечують повноцінну взаємодію користувача із застосунком.

У ході роботи реалізовано механізми реєстрації та авторизації користувачів, головну сторінку з відображенням зображень, логіку додавання та перегляду цифрових зображень, роботу з альбомами, коментарями та рейтингами, а також управління особистими даними користувача. Окрему увагу приділено реалізації серверних API-ендпоінтів для збереження, обробки та отримання даних із бази SQLite.

Результати третього розділу підтверджують коректність реалізації основного функціоналу мобільного застосунку та його готовність до подальшого тестування, розширення можливостей і практичного використання для організації та управління цифровими зображеннями.

ВИСНОВОК

У процесі виконання дипломної роботи було проведено аналіз підходів до організації та керування цифровими зображеннями у мобільних застосунках. Огляд наявних рішень дозволив визначити основні потреби користувачів, зокрема: зручне завантаження і перегляд зображень, їх структурування за допомогою альбомів і тегів, керування рівнями доступу, а також можливість взаємодії з контентом через коментарі та оцінювання. На основі отриманих результатів було сформульовано функціональні та технічні вимоги до архітектури застосунку, організації бази даних і взаємодії клієнтської та серверної частин.

У ході розробки мобільного застосунку реалізовано базовий функціонал керування цифровими зображеннями, який включає реєстрацію та авторизацію користувачів, завантаження зображень із метаданими, формування персональної галереї, перегляд детальної інформації про зображення, а також роботу з альбомами. Додатково реалізовано механізми взаємодії з контентом: коментування, оцінювання, підрахунок переглядів, завантаження зображень та керування їх приватністю. Передбачено сторінку профілю користувача з можливістю редагування особистих даних і перегляду статистики активності. Клієнтську частину реалізовано з використанням React Native, серверну - на базі Node.js, а зберігання даних організовано у реляційній базі SQLite.

Проведене тестування підтвердило коректність роботи реалізованого функціоналу, стабільність обміну даними між клієнтською та серверною частинами, а також відповідність інтерфейсу вимогам зручності для мобільних пристроїв. Отримані результати свідчать про можливість подальшого розвитку застосунку шляхом розширення функціональних можливостей, оптимізації роботи з зображеннями та впровадження додаткових сервісів, що підвищать практичну цінність розробленої системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Gonzalez R. C., Woods R. E. Digital Image Processing. 4th ed. Boston : Pearson, 2018. 1168 p.

2. Burger W., Burge M. J. Digital Image Processing: An Algorithmic Introduction Using Java. 2nd ed. Cham : Springer, 2016. 811 p.

3. Pratt W. K. Digital Image Processing: PIKS Scientific Inside. 4th ed. Hoboken : John Wiley & Sons, 2007. 807 p.

4. Sonka M., Hlavac V., Boyle R. Image Processing, Analysis, and Machine Vision. 4th ed. Boston : Cengage Learning, 2014. 944 p.

5. Canny J. A Computational Approach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence. 1986. Vol. 8, No. 6. P. 679–698.

6. Otsu N. A Threshold Selection Method from Gray-Level Histograms. IEEE Transactions on Systems, Man, and Cybernetics. 1979. Vol. 9, No. 1. P. 62–66.

7. Wang Z., Bovik A. C., Sheikh H. R., Simoncelli E. P. Image Quality Assessment: From Error Visibility to Structural Similarity. IEEE Transactions on Image Processing. 2004. Vol. 13, No. 4. P. 600–612.

8. Flanagan D. JavaScript: The Definitive Guide. 7th ed. Sebastopol : O'Reilly Media, 2020. 706 p.

9. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. 2nd ed. Sebastopol : O'Reilly Media, 2020. 310 p.

10. Eisenman B. Learning React Native: Building Native Mobile Apps with JavaScript. 2nd ed. Sebastopol : O'Reilly Media, 2017. 298 p.

11. Subramanian V. Pro MERN Stack: Full Stack Web Development with Mongo, Express, React, and Node. 2nd ed. Berkeley : Apress, 2019. 550 p.

12. Gackenheimer C. Introduction to React Native. Berkeley : Apress, 2015. 219 p.

13. Wieruch R. The Road to React: Your Journey to Master Plain yet Pragmatic React.js. Independently published, 2022. 250 p.

14. Resig J., Bibeault B., Maras J. Secrets of the JavaScript Ninja. 2nd ed. Shelter Island : Manning Publications, 2016. 464 p.

15. Ducker B. Express.js: Guide Book. [Б. м.] : Independently published, 2019. 242 p.

16. Kauffman C. SQLite Tutorial: Create, Manage and Access SQLite Databases. [Б. м.] : Independently published, 2020. 310 p.

17. Eisenberg J. D. SVG Essentials: Producing Scalable Vector Graphics with XML. 2nd ed. Sebastopol : O'Reilly Media, 2019. 310 p.

18. Lowe D. G. Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision. 2004. Vol. 60, No. 2. P. 91–110.

19. Mozilla Developer Network. JavaScript documentation. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (date of access: 09.05.2026).

20. React Native Documentation. URL: https://reactnative.dev/docs/getting-started (date of access: 09.05.2026).

21. Goode C. Full Stack JavaScript: Learn Backbone.js, Node.js, and MongoDB. [Б. м.] : Apress, 2018. 230 p.

22. Lyon B. Modern Full-Stack Development: Using TypeScript, React, Node.js, Webpack, and Docker. [Б. м.] : Apress, 2020. 300 p.

23. Ogundipe O. Securing REST APIs with Node.js: Authentication and Authorization with JWT. [Б. м.] : Independently published, 2021. 180 p.

24. Chen Y., Zhang Y., Liu X. Deep Learning for Image Retrieval: A Survey. IEEE Access. 2021. Vol. 9. P. 21068–21088.

25. Wilson J. Progressive Web Apps: The Definitive Guide. Sebastopol : O'Reilly Media, 2019. 250